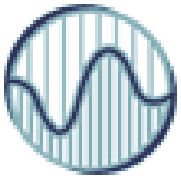
Горан Дикић

Слободан Драшковић

МИКРОРАЧУНАРИ

~прво издање~



ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА СТРУКОВНИХ СТУДИЈА, БЕОГРАД

АКАДЕМИЈА ТЕХНИЧКО-УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА, БЕОГРАД

2020

Аутори: *др Горан Дикић, Слободан Драшковић маст. инж.*

Рецензенти:

др Милан Мијалковић, професор Високе школе електротехнике и рачунарства струковних студија у Београду

др Перица Штрбац, професор Високе школе електротехнике и рачунарства струковних студија у Београду

Издавачи:

Висока школа електротехнике и рачунарства струковних студија, Београд

Академија техничко-уметничких струковних студија, Београд

Обрада текста:

др Горан Дикић, Слободан Драшковић маст. инж.

Корице:

Ненад Селић спец. струк. инж.

Тираж: 30

Штампа: Развојно-истраживачки центар графичког инжењерства ТМФ, Београд

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд

004.382.7(075.8)
004.31(075.8)

ДИКИЋ, Горан, 1960-
Микрорачунари / Горан Дикић, Слободан Драшковић. - 1. изд. - Београд :Висока школа електротехнике и рачунарства струковних студија: Академијатехничко-уметничких струковних студија, 2020 (Београд:Развојно-истраживачки центар графичког инжењерства ТМФ). - 212 стр. :илустр. ; 24 cm

Тираж 30. - Библиографија: стр. 209. - Регистар.

ISBN 978-86-7982-327-4 (ВШЕРСС)

1. Драшковић, Слободан, 1989- [аутор]
a) Рачунари b) Микропроцесори
COBISS.SR-ID 22383625

Предговор

Уџбеник је намењен студентима основних струковних студија који у свом студијском програму изучавају предмет Микрорачунари. Садржај је структуиран у дванаест поглавља која се излажу у истоветном редоследу током предавања. Редослед тема обезбеђује увођење студената у основе програмирања микрорачунара полазећи од упознавања основних појмова па све до проблема оптимизације програма у погледу захтева за рад у реланом времену. Саставни део текста чине примери програма који су развијени користећи развојне системе познате под називом ARDUINO и засновани су на осмобитном микроконтролеру ATmega328/P. Уз овај уџбеник студентима је на располагању и практикум за вежбе.

У првом поглављу се описују основна структура рачунара, улога његових основних елемената као и њихова међусобна повезаност. Поред тога дефинишу се узајамни односи између појмова као што су микропроцесор, рачунар и микрорачунар.

У другом поглављу приказана је основна архитектура микроконтролера. Упоређене су RISC и CISC архитектура. Описана је суштина пајпланј концепта и његов значај за обраду података у реалном времену. Дат је приказ основних програмских језика као што су машински језик, асемблер и виши програмски језици.

Треће поглавље започиње описом општих градивних блокова већине програмских језика. Дефинишу се основи програмски кораци које треба имати у виду при развоју програмске подршке за решење неког конкретног проблема. На крају је дат приказ програмског окружења у којем се развија програм за системе типа ARDUINO.

Четврто поглавље представља приказ типова података, поступак дефинисања и декларисања

променљивих као и конверзију података из једног типа у други.

Пето поглавље садржи опис оператора који се користе у развоју програмске подршке, њихове међусобне односе у погледу приоритета као и примере за илустрацију њиховог коришћења.

Пето поглавље започиње навођењем основних управљачких структура након чега следи детаљан опис сваке од њих. Поред тога дати су и примери којим се илуструје начин коришћења као и ефекти примене одговарајуће структуре на ток програма.

У седмом поглављу је описан поступак аналогно-дигиталне конверзије који се заснива на методу сукцесивне апроксимације. С обзиром да конкретан микроконтролер не садржи дигитално-аналогни конвертор описан је поступак апроиксимативног приступа у поступку добијања аналогног сигнала који се заснива на такозваној ширинско пулсној модулацији (PWM). Цео поступак је илустрован приказом одговарјућег примера.

У осмом поглављу су дефинисани основни поступци за асинхрони и синхрони пренос података. Дефинисани су појмови као што су бодска брзина и формат податка. Поред тога описани су и начини повезивања уређаја.

У деветом поглављу описано је како се развија функција. Дефинисани су појмови као што су спецификација и тип функције, аргументи, тело функције и њен потпис.

У десетом поглављу је дефинисана употреба прекида. Описан је поступак обраде спољног прекида. Поред тога дат је пример којим се илуструје развој функцијског програма за обраду прекида.

У једанаестом поглављу је описана суштина показивача, односно како се дефинишу и начин њиховог коришћења. Посебан нагласак је усмерен на проблем

дефинисања вредности показивача, дефинисање вредности променљиве помоћу показивача и пренос више података помоћу показивача.

У последњем поглављу, централно питање, представља проблем оптимизације како хардверских тако и софтверских ресурса у складу са захтевима обраде података у реалном времену.

Аутори се захваљују рецензентима, професору др Милану Мијалковићу и професору др Перици Штрбцу на сугестијама које су допринеле да садржај уџбеника буде бољи.

У Београду, септембар 2020. год.
Аутори

САДРЖАЈ

1

УВОД У МИКРОРАЧУНАРЕ

У овом поглављу описују се основна структура рачунара, улога његових основних елемената као и њихова међусобна повезаност. Поред тога дефинишу се узајамни односи између појмова као што су микропроцесор, рачунар и микрорачунар.

Први уређаји за обављање рачунских операција су развијани на бази механичких компоненти. Напредак у области електронике је омогућио развој аналогних рачунара који су се користили у истраживачким лабораторијама и ситемима аутоматског управљања. Прави напредак у смислу шире примене рачунара везује се за развој полупроводничке технологије. Производња дигиталних интегрисаних кола омогућила је развој брзих, поузданих уређаја који су могли не само да обаве рачунске операције већ и да ускладиште огромне количине информација. Упоредо са развојем дигиталних рачунара унапређене су и комуникације тако да се све више користе дигитални системи за пренос информација. Управо смо сведоци својеврсне револуције која је довела до тога да је аналогни пренос TV сигнала замењен дигиталним.

Чување, обрада и пренос информација у дигиталном облику срећу се у бројним апликацијама, укључујући контролу процеса, комуникационе системе, дигиталне инструменте и производе широке потрошње. Дигитални рачунар, који се најчешће скраћено назива рачунар, представља пример типичног дигиталног система.

Рачунар, користи информације у дигиталном, или прецизније речено, бинарном облику. Бинарни број има само две дискретне вредности - нула или јединица. Свака од ових дискретних вредности представља заправо једно од стања електронског прекидача који је реализован помоћу транзистора (0 – транзистор проводи струју, 1 – транзистор не проводи струју). Дакле, сви рачунари „разумеју” само бинарне бројеве. Сваки децимални број (база 10, са десет цифара од 0 до 9) може да се представи бинарним бројем (база 2, са цифрама 0 и 1).

Основни блокови рачунара су централна процесорска јединица (Central Processing Unit - CPU*),* меморија и улазно/излазни уређаји *(*Input/Output - I/O). CPU рачунара у

суштини има сличну улогу као и мозак човека. Рачунарска меморија је концептуално слична људској меморији. Питање које се упућује човеку одговара уласку у програм рачунара користећи улазни уређај као што је тастатура. С друге стране, одговор човека на то питање концепцијски одговара резултату који се појављује на излазном уређају рачунара, као што су екран или штампач. При томе треба имати у виду да људска бића могу самостално да размишљају, док рачунари могу да одговоре само на питања за која су програмирани. Рачунарски хардвер представљају компоненте рачунара као што су меморија, CPU, транзистори и механичке компоненте неопходне за њихов смештај у јединствену целину. Програми могу да изврше одређени задатак, као што је сабирање, ако рачунар садржи електронско коло које може да сабере два броја. Програмери не могу да мењају ова електронска кола, али могу да обављају задатке на њима користећи инструкције односно наредбе.

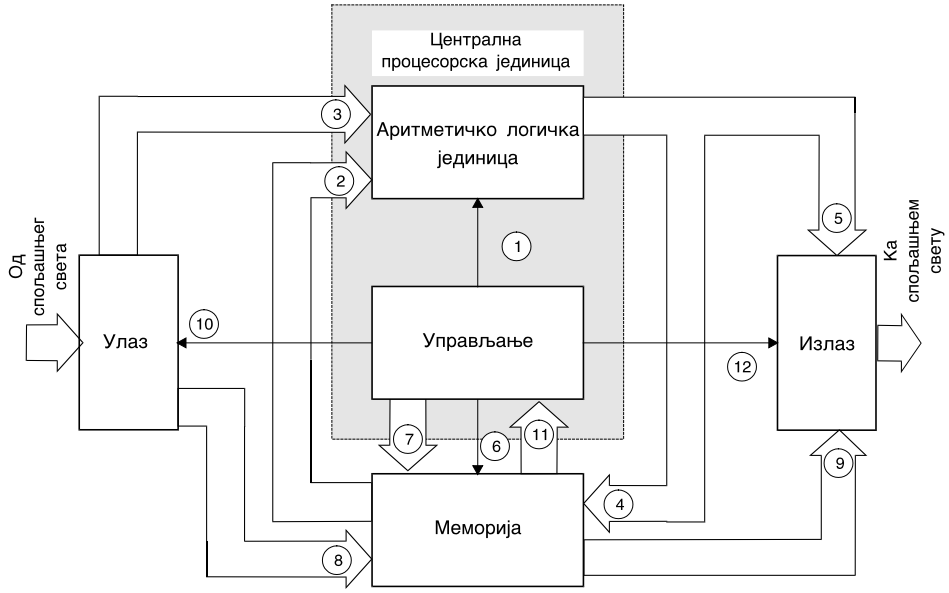
Напретком у области полупроводничке технологије, произведена је централна процесорска јединица CPU у једном интегрисаном колу, позната под називом микропроцесор. Упоредо са развојем микропроцесора, развијане су и одговарајуће меморије као и интегрисана кола за подршку улазно излазних процеса, а све са намером да се произведе јединствен уређај познат под називом микрорачунар. Прикључци (пинови) на сваком од ових чипова могу да се повежу са одговарајућим линијама на системској сабирници, која се састоји од адреса, података и управљачких линија. У прошлости су поједини произвођачи дизајнирали комплетан микрорачунар, са ограниченим могућностима, на једном интегрисаном колу. Такви микрорачунари су познати под називом микроконтролери и обично се користе за наменске апликације као што су аутомобилски системи, кућни апарати и системи за кућну забаву. Типични микроконтролер, садржи

микрорачунар, тајмере, AD (аналогно-дигитални) и DA (дигитално-аналогни) конвертор.

1.1. Основна структура рачунара

Без обзира на разлике у перформансама рачунара сваки од њих има пет основних јединица: аритметичко-логичку јединицу ALU, меморијску јединицу, управљачку јединицу, улазну и излазну јединицу. Слика 1.1 приказује блок дијаграм који представља општи рачунарски систем и показује везу различитих јединица (блокова).

Уочимо да је управљачка јединица центар дијаграма пошто функционише као мозак рачунара, управљајући радом свих других блокова. Стрелице у дијаграму показују правац у коме теку подаци или управљачки сигнали. Велика стрелица представља групу паралелних линија које преносе реч података, инструкцију или адресу; мања стрелица представља управљачке сигнале, који су обично једна или неколико сигналних линија. У даљем тексту користићемо уоквирене бројеве поред стрелица ради описа сваке од јединица.



Слика 1.1 Општи блок дијаграм рачунарског система

1.1.1. Меморијска јединица

У овој јединици смештене су инструкције у виду група битова које рачунар треба да изврши (на пример програм), и податке са којима треба радити у програму. Меморија служи такође и као привремено складиште резултата операција које извршава ALU (Aritmetic Logic Unit - стрелица 4).

Меморијском јединицом управља управљачка јединица, која сигнализира операције очитавање (READ) или уписивање (WRITE) (стрелица 6) и обезбеђује одговарајућу меморијску адресу (стрелица 7).

Подаци који треба да се упишу у меморију могу да дођу из ALU или из улазне јединице (стрелица 8), а све под контролом управљачке јединице. Подаци који су учитани из меморије могу да се шаљу ка ALU (стрелица 2) или ка излазној јединици (стрелица 9).

Као што је већ напоменуто рачунар има интерну и екстерну меморију. Интерна меморија је релативно малог капацитета, брза, и смешта програме и податке које рачунар тренутно извршава (краткотрајно смештање). Полупроводнички RAM (Random Access Memory) и ROM (Read Only Memory) се обично користе као интерна меморија. Екстерна меморија омогућава смештање великих програма и бројних података које рачунар тренутно не користи (смештање на дужи период). Магнетне траке и дискови су, у прошлости, били уобичајни типови екстерне меморије.

1.1.2. Аритметичко логичка јединица

Аритметичко-логичка јединица, ALU, обавља аритметичке и логичке операције над подацима. Управљачка јединица шаље сигнале ка ALU како би се дефинисале операције које ће се извршавати над подацима (стрелица 1). Управљачка јединица одређује и извор података - меморијску јединицу (стрелица 2) или улазну јединицу (стрелица 3). Резултат операције може бити послат у меморију ради ускладиштења (стрелица 4) или ка спољашњој јединици (стрелица 5).

1.1.3. Улазна јединица

Улазна јединица се састоји од склопова који омогућавају да подаци и информације који долазе споља улазе у интерну меморију рачунара или ALU (стрелице 3 и 8). Ови склопови се обично називају периферије пошто су они физички одвојени од склопова који чине мозак рачунара. Првобитни рачунари су користили улазне периферије у виду читача бушених картица, читача папирне траке, тастатура, телепринтера и аналогно дигиталних конвертора (ADC). Персонални рачунари имају тастатуру као улазни уређај.

Магнетна трака и диск јединице често су посматране као улазне периферије. Њихова главна функција је да сместе

податке и програме који се, после команди из управљачке јединице, преносе у рачунарску интерну меморију.

1.1.4. Излазна јединица

Излазна јединица се састоји од периферијских уређаја који преносе податке и информације из интерне меморије или ALU ка спољашњем свету (стрелице 5 и 9). Типичне излазне периферије укључују LED дисплеј, штампаче, видео мониторе, и дигитално-аналогне конверторе (DAC).

Магнентне траке и диск меморијске јединице могу се сматрати и као излазне периферије (јединице масовне меморије); подаци и програм из интерне меморије смештају се у једној од спољашњих меморија на трајно чување.

Терминал је тастатура (као улазна јединица) комбинована са јединицом за визуелно приказивање (као што су штампачи или видео монитори). Комбинација тастатура/видео показивач (дисплеј) назива се видео терминал. Микрорачунар обично има један терминал, док већина великих рачунара и многи минирачунари имају неколико терминала.

1.1.5. Управљачка јединица

Ова јединица управља радом свих других делова рачунара обезбеђујући временске и управљачке сигнале. Она је као диригент у оркестру који води рачуна да је сваки од инструмената у доброј синхронизацији. Управљачка јединица узима инструкцију из меморије тако што шаље адресу (стрелица 7) и команду за очитавање (стрелица 6) ка меморијској јединици. Инструкција (реч) из меморије долази у управљачку јединицу (стрелица 11). С обзиром да је инструкцијска реч у бинарном кодираном облику неопходно је да се најпре декодује (преко логичких кола у управљачкој јединици) како би се одредило која операција треба да се

изводи. Када се операције одреде, управљачка јединица генерише секвенцу сигнала који је извршавају. Очигледно, управљачка јединица има задатак да узме, декодује и изврши инструкције које су у меморији (тзв. програм).

1.1.6. Централна процесорска јединица (CPU)

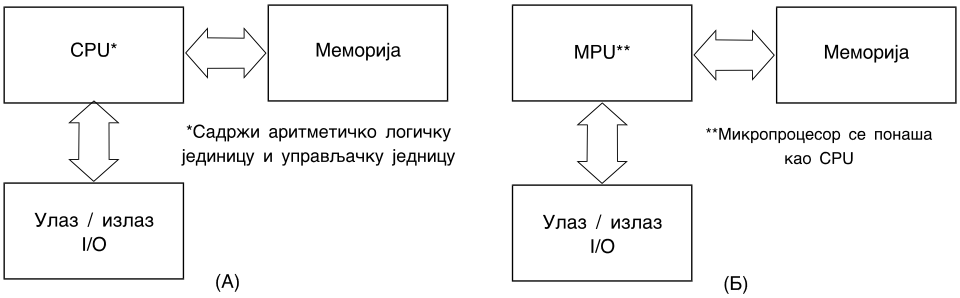
ALU и управљачка јединица обично су повезани у једну јединицу која се назива централна процесорска јединица CPU (Central processing Unit). CPU је заиста "мозак" рачунара јер садржи кола која генеришу све управљачке сигнале и потребни су за извршавање инструкција са колима која извршавају операције у складу са инструкцијским захтевима.

1.2. Микрорачунар

У микрорачунару, CPU је интегрисано коло које се назива микропроцесор. Термини "CPU" и "микропроцесор" се често користе као синоними. У рачунарству је уобичајно да се микропроцесор назива микропроцесорска јединица - MPU (MicroProcessor Unit).

1.3. Поједностављен дијаграм рачунарског система

Пошто смо се упознали са функцијама пет основних јединица рачунарског система, можемо комбиновати неке од јединица да би добили једноставнији дијаграм. Слика 1.2 показује како се рачунар може представити као три блока. CPU блок замењује ALU и управљачку јединицу. Улазно излазни блок (И/О) је комбинација улазних и излазних јединица.



Слика 1.2 (А) Поједностављени дијаграм рачунара; (Б) Дијаграм микрорачунара

1.4. Сажетак

Основни блокови рачунара су централна процесорска јединица (Central Processing Unit - CPU), меморија и улазно/излазни уређаји (Input/Output - I/O). CPU је комбинација ALU и управљачке јединице било ког рачунара. Када је CPU једно интегрисано коло назива се микропроцесор и означавамо га са MPU. У случају када је MPU повезан са меморијом и I/O уређајима добија се конфигурација која је позната као микрорачунар. Обједињавањем микропроцесорске јединице, меморије и периферних уређаја као што су тајмери, комуникациони подсистеми за серијски пренос података, AD и DA конвертори направљена су специфична интегрисана кола позната под називом микроконтролери. У зависности од њихове сложености могуће је у потпуности реализовати управљачке системе за потребе аутомобилске индустрије, сигурносних система, комуникационих модема, подсистема за одржавање температуре и притсика, уређаја за домаћинство, и слично.

Питања

1. Објасните у чему се разликују микропроцесор, микрорачунар и микроконтролер.
2. Шта чини основну структуру микрорачунара?
3. Објасните интеракцију управљачке јединице са осталим деловима рачунара.
4. Наведите типичне примере улазних јединица.
5. Наведите типичне примере излазних јединица.
6. Шта чини централну процесорску јединицу?

2

ОСНОВЕ МИКРОКОНТРОЛЕРА

У овом поглављу приказана је основна архитектура микроконтролера. Упоређене су RISC и CISC архитектура. Поред тога описана је суштина пајпланј концепта и његов значај за обраду података у реалном времену. Дат је приказ основних програмских језика као што су машински језик, асемблер и виши програмски језици.

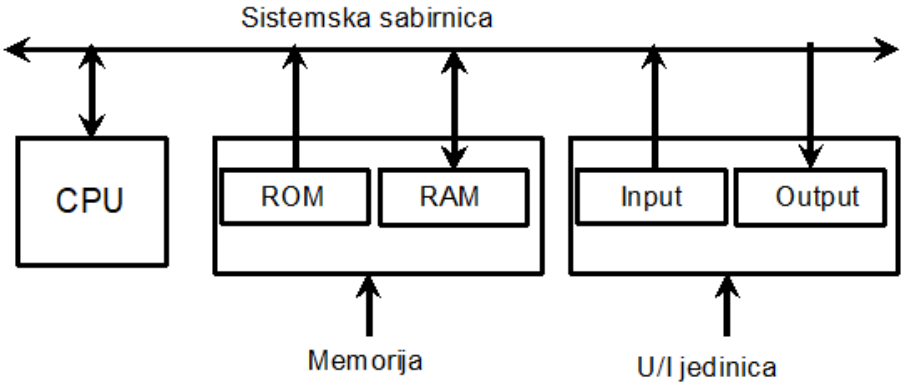
У тексту који следи описани су појмови који су неопходни за разумевање основних карактеристика микроконтролера. Приказане су типичне архитектуре микроконтролера (на нивоу блок дијаграма), њихове предности и мане, временски сигнали као и пајплајн организација извршења инструкција.

2.1. Основни блокови микроконтролера

Да би се разумеле функције које обављају типични модули садржани у микроконтролеру, потребно је упознати основне блокове микрорачунара.

Микрорачунар има три основна блока: микропроцесор (Central Procesor Unit - CPU на чипу), меморијску јединицу и улазно/излазну (input/output - I/O) јединицу. На слици 2.1 приказани су основни блокови микрорачунара. Системска магистрала (системска сабирница) повезује ове блокове. CPU извршава све инструкције и обавља аритметичке и логичке операције над подацима. CPU микрорачунара садржи све регистре и контролну јединицу, као и аритметичко-логичка кола микрорачунара.

Меморијска јединица (*memory unit*) чува и податке и програм са инструкцијама. Одељак меморије обично садржи ROM (*Read Only Memory*) и RAM (*Random Accses Memory*) интегрисана кола. ROM се може само читати и неизбрисив је, односно, задржава свој садржај када је напајање искључено. ROM се обично користи за чување инструкција и података који се не мењају. На пример, може да чува табелу кодова за приказ података (цифре од 0 до 9) на седмосегментном дисплеју.



Слика 2.1 Основни блокови микро рачунара

За разлику од ROM-а RAM може да се исчитава и уписује. RAM је избрисив, односно, не задржава свој садржај када је напајање искључено. RAM се користи за чување инструкција и података који су привремени и могу се мењати током извршавања програма. Улазно-излазна (I/O) јединица преноси податке између микрорачунара и спољних уређаја преко I/O портова (регистара).

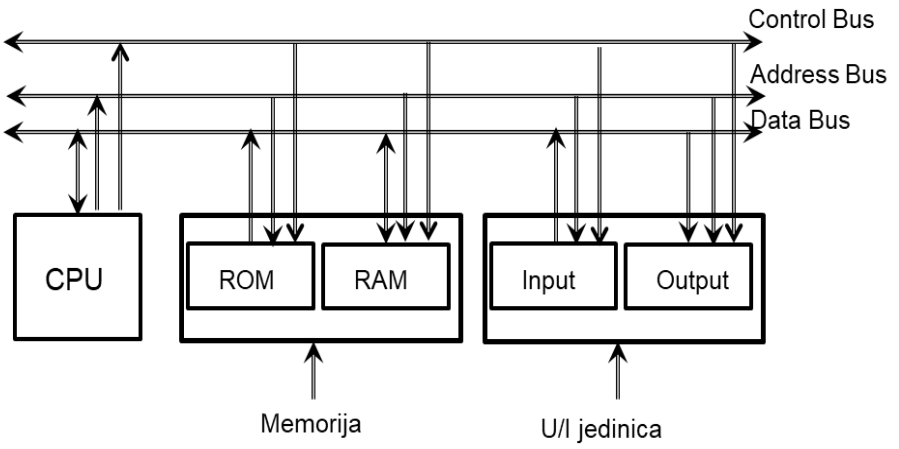
Микроконтролери садрже CPU, меморију и IOP (I/O и Периферије) на једном интегрисаном колу. Имајте на уму да типични IOP садржи I/O јединицу микрорачунара, тајмере, AD (аналогно-дигиталне) претвараче, аналогне компараторе, серијске I/O и друге периферне функције (о чему ће бити више речи у тексту који следи).

Слика 2.2 илуструје веома поједностављену верзију типичног микрорачунара и приказује основне блокове микрорачунарског система као и магистрале које повезују ове блокове. Иако ова слика изгледа веома једноставно, она укључује све главне елементе типичног микрорачунарског система.

2.1.1. *Системска магистрала (System Bus)*

Системска магистрала микрорачунара (унутрашња за микроконтролер) садржи три магистрале, које преносе све адресе, податке и контролне информације укључене у извршење програма. Ове магистрале повезују CPU са сваким ROM-ом, RAM-ом и I/O уређајиматако да је могуће пренети информације између CPU-а и било којег другог елемента. У микрорачунару се типични пренос података врши у односу на меморију или I/O. Процес у току којег меморијска јединица или I/O уређај прима податке од CPU-а, назива се операција уписа (WRITE operation). Подаци, након овог процеса, остају уписани у изабрану меморијску локацију или I/O порт (регистар). Процес у току којег меморијска или I/O јединица шаље податке ка CPU-у, назива се операција читања (READ *operation*). Подаци су, након овог процеса, очитани у CPU са изабране меморијске локације или I/O порта.

На адресној магистрали (*address bus*), пренос информација се одвија у само једном правцу (једносмерна магистрала - *unidirectional bus*), од CPU-а ка меморији или I/O јединици.



Слика 2.2 Поједностављена верзија типичног микрорачунара.

Величина адресне магистрале одређује укупан број доступних меморијских адреса у оквиру којих могу да се извршавају програми применом конкретног микропроцесора. Адресна магистрала се димензионира у складу са укупним бројем адресних бита конкретне CPU. Ово такође одређује могућност непосредног адресирања или величину главне меморије микроконтролера. CPU микроконтролера може да извршава само програме који се налазе у главној меморији. На пример, CPU са 16 адресних битова може да адресира 2^{16} = 65.536 бајта [64 килобајта (Kb)] различитих могућих адреса (комбинација сачињених од јединица и нула) на адресној магистрали. CPU укључује адресе од 0 до $65{,}535_{10}$ (0000_{16}-$FFFF_{16}$). Свака меморијска локација може бити представљена једном од ових адреса. На пример, 8-битни податак $2B_{16}$ може бити сачуван на 16-битној адреси 0200_{16}.

Када CPU са 16-битном адресном магистралом треба да пренесе информације између себе и одређене меморијске локације, она генерише 16-битну адресу из унутрашњег регистра на својих 16 адресних прикључака, A0-A15, која се затим појављује на адресној магистрали. Ове 16-битне адресе се декодирају како би се одредила жељена локација меморије. Процес декодирања обично захтева хардвер (декодере) који нису приказани на слици 2.2. На магистрали података (*data bus*), подаци могу да струје у оба смера, односно ка или од CPU-а. Због тога је ово двосмерна магистрала.

Контролна магистрала се састоји од више сигнала који се користе за синхронизацију рада појединих микрорачунарских елемената. CPU шаље неке од ових контролних сигнала другим елементима како би указао на врсту извршене операције. Сваки микроконтролер има јединствени скуп контролних сигнала. Међутим, неки

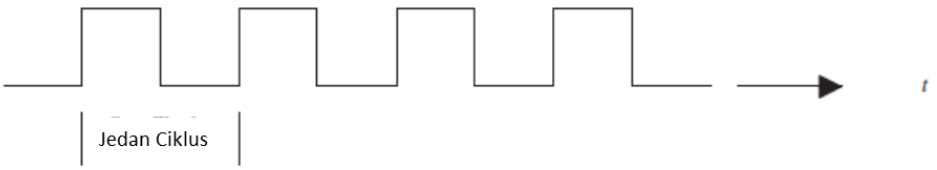
контролни сигнали као што су READ и WRITE су уобичајени за све микроконтролере.

2.1.2. Сигнали системског такта (Clock сигнали)

Сигнали системског сата се преносе у оквиру контролне магистрале. Ови сигнали генеришу одговарајуће периоде током којих CPU извршава инструкције. Типични микроконтролери поседују интерно електрично коло (генератор такта) које генерише сигнал системског такта (*Clock signal*). Слика 2.3 приказује типичан облик овог сигнала.

Број циклуса у секунди (Hertz, скраћено Hz) назива се фреквенција такта. Фреквенција CPU такта, у случају типичних микроконтролера, варира од 1MHz ($1 \cdot 10^6$Hz) до више стотина MHz. Генератор такта дефинише брзину микроконтролера. Имајте на уму да је трајање једног циклуса такта = 1/*f*, где је *f* његова фреквенција. Времена извршења инструкција микроконтролера обично се дефинишу у виду броја циклуса такта.

Претпоставимо, на пример, да је време извршења инструкције додавања (ADD) од стране микроконтролера један циклус. То значи да ће микроконтролер са системским сатом који ради на фреквенцији од 40MHz извршити ADD инструкцију за 25 наносекунди [циклус такта = $1/(40 \cdot 10^6$Hz) = 25 наносекунди]. С друге стране, за микроконтролер са системским тактом који ради на фреквенцији од 4MHz, ова инструкција ће бити извршена за 250 наносекунди [циклус такта = $1/(40 \cdot 10^6$Hz) = 250 наносекунди]. Очигледно, микроконтролер брже извршава инструкције уколико је већа фреквенција такта.



Slika 2.3 Типичан облик сигнала системског такта (*Clock signal*)

2.2. Архитектуре микроконтролера

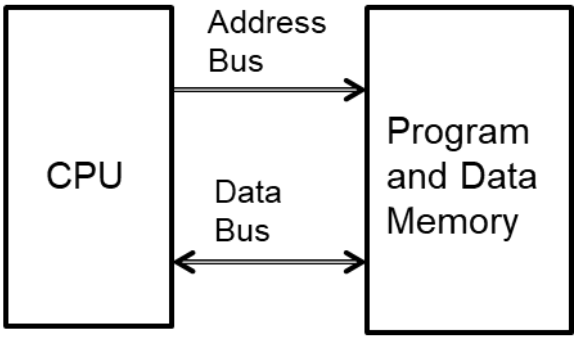
Микроконтролер захтева меморију за складиштење програма и података. Различити микроконтролери који су доступни данас у принципу су исти. Главне варијације су у броју меморијских јединица, као и адреса и магистрала података које користе. Као што је поменуто у Поглављу 1, за дизајнирање микроконтролера се користе две врсте архитектура. То су фон Нојман (*Princeton*) и Харвард.

У архитектури фон Нојман, са једним меморијским системом, користи се иста адресна магистрала и магистрала података било да се приступа програмима или подацима. То значи да се програмима (инструкцијама) и подацима не може приступити истовремено. Ово може смањити укупну брзину. Слика 2.4 приказује блок дијаграм фон Нојман архитектуре.

Харвард архитектура микроконтролера користи засебне јединице за смештај програма и чување података упоредо са засебним магистралама за инструкције и податке. То значи да ови микроконтролери могу истовремено извршавати инструкције и приступати подацима. Микроконтролери дизајнирани са овом архитектуром захтевају четири магистрале за програмску меморију и меморију података. То су: једна магистрала података за инструкције, једна адресна магистрала за адресе инструкција, једна магистрала података за податке и једна адресна магистрала за адресе података. Величине адресне магистрале и магистрале података за инструкције могу се разликовати од величина адресне

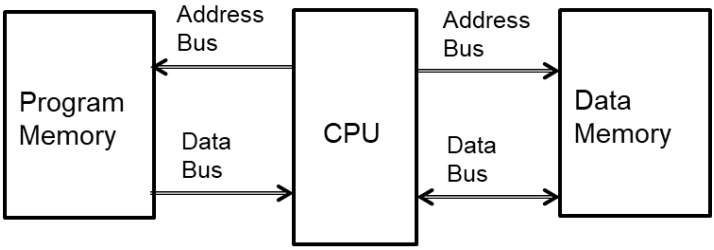
магистрале и магистрале података за податке. Слика 2.5 приказује блок дијаграм ове архитектуре.

Већина микроконтролера користи Харвард архитектуру. Разлог за то је јефтина имплементација ових магистрала унутар чипа јер су меморије програма и података смештене унутар њега.



Слика 2.4 фон Нојман архитектура

Иако су процесори дизајнирани коришћењем архитектуре фон Нојман спорији у поређењу са Харвард архитектуром (инструкције и подаци нису доступни истовремено због јединствених магистрала), типични микропроцесори као што је Пентиум користе ову архитектуру. То је зато што су меморијске јединице као што су ROM и RAM спољне у односу на микропроцесор. Овај приступ захтева готово половину броја жица на матичној плочи, јер се адресе и подаци преносе путем само две магистрале, а не четири као у случају Харвард архитектуре. То је разлог зашто би Харвард архитектура била веома скупа ако се користи за дизајнирање микропроцесора.



Слика 2.5 Харвард архитектура

Коначно, у зависности од регистарског одељка, може се рећи да постоје CPU чији се рад заснива на примени акумулатора и CPU чији се рад заснива на примени регистара опште намене. У случају CPU чији се рад заснива на примени акумулатора подаци који се појављују као резултат обраде су увек смештени у регистру званом акумулатор. Све аритметичке и логичке операције се извршавају помоћу акумулатора као једног од извора података. С друге стране, у случају CPU чији се рад заснива на примени регистара опште намене, било који регистар опште намене може се користити као акумулатор.

2.3. Основе проточне обраде

Бројни микроконтролери заснивају свој рад, између осталог, на примени проточне обраде (пајплајн). Као што је познато, током извршења програма, стандардни CPU понавља следећа три корака за комплетирање сваке инструкције:

1. Дохватање (Fetch). CPU преузима инструкције из меморије програма (спољне према CPU-у) у регистар инструкција.

2. Декодирање (Decode). CPU декодира или преводи инструкције помоћу контролне јединице. Контролна јединица уноси садржај инструкцијског регистра, а затим декодира (преводи) инструкцију да одреди тип инструкције.

3. Изврши (Execute). CPU извршава инструкцију помоћу контролне јединице. Да би извршио задатак, управљачка јединица генерише одговарјуће сигнале у складу са захтевима инструкција.

Претпоставимо, на пример, да је потребно збројити садржај два регистра, X и Y и чувати резултат у регистру Z. Да би се ово постигло, конвенционални CPU извршава следеће кораке:

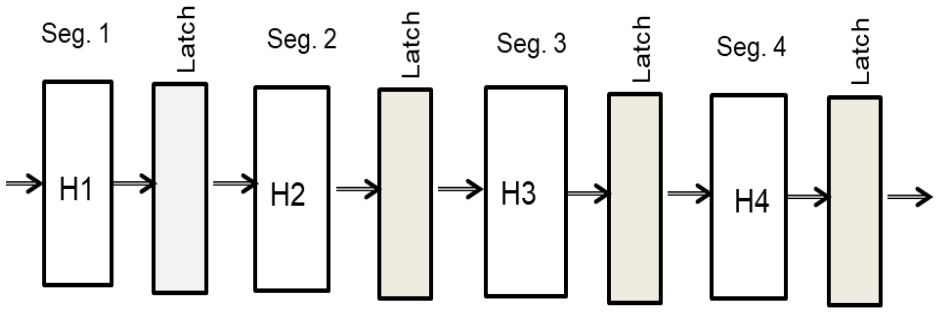
1. CPU дохвата инструкцију и смешта у инструкцијски регистар.
2. Контролна јединица (Control Unit - CU) декодира садржај инструкцијског регистра.
3. CU извршава инструкцију генеришући (ENABLE) сигнале који омогућавају интеракцију регистара и ALU-а да изврше следеће:
   - CU преноси садржаје регистара X и Y из регистарске секције у ALU.
   - CU захтева од ALU да изврши збрајање ових садржаја.
   - CU преноси резултат из ALU у Z регистар (унутар регистарске секције).

Основни појмови повезани са проточном обрадом биће размотрени у тексту који следи. Претпоставимо да се задатак T извршава обављањем четири активности: A_1, A_2, A_3 и A_4, у том редоследу. Хардвер H_i је дизајниран да обавља активност A_i и назива се сегмент (*i=1,2 ,3...*). У суштини то је електронско коло које садржи елементе комбинационих кола. Размотримо конфигурацију која је приказан на слици 2.6.

У овој конфигурацији, између два сегмента, поставља се привремена меморија (*latch*), тако да резултат израчунат од стране једног сегмента може послужити као улаз за следећи сегмент током наредног временског периода. Извођење четири задатка T1, T2, T3 и T4 помоћу хардвера на слици 2.6

описано је коришћењем просторно-временског графикона који је приказан на слици 2.7.

У почетку се задатак Т1 подржава сегментом 1. Након првог временског периода (трајање је одређено фреквенцијом клок сигнала), сегмент 2 је заузет обрадом задатка Т1 док је сегмент 1 заузет са Т2. Настављајући на овај начин, задатак Т1 је завршен на крају четвртог временског периода. Међутим, након ове тачке, један задатак је у потпуности обављен и резултат се испоручује у сваком наредном временском периоду. Ово је суштина проточне обраде (пајплајн приступа). Ефикасност овог приступа је заснована на истом принципу који се користи на модерним прозводним линијама (неколико активности се обављају истовремено, али не и на истом материјалу).



Слика 2.6 Четворосегментна пајплајн организација

Завршен					T1	T2	T3	T4
Seg. 4				T1	T2	T3	T4	
Seg. 3			T1	T2	T3	T4		
Seg. 2		T1	T2	T3	T4			
Seg. 1	T1	T2	T3	T4				
Циклус	1.	2.	3.	4.	5.	6.	7.	8.

Слика 2.7 Преклапање извршења четири задатка помоћу пајплајн организације извршења инструкција.

2.4. Поређење RISC и CISC архитектура

Постоје два типа CPU архитектуре: RISC (*Reduced Instruction Set Computer* - смањени инструкцијски сет) и CISC (*Complex Instruction Set Computer* - комплексни компјутерски сет инструкција). RISC микроконтролер је дизајниран тако да у први план поставља захтеве као што су једноставност и ефикасност. RISC дизајн започиње са неопходним и довољним инструкцијама. Коришћење RISC архитектуре треба да максимизира брзину обраде смањењем броја такт циклуса по инструкцији. Скоро сва израчунавања могу се обавити унутар неколико једноставних операција. Циљ RISC архитектуре је да максимизира ефективну брзину обављањем ретких операција у софтверу (реализују се у виду низа инструкција) и честим функцијама у хардверу (реализују се применом одговарајућег компбинационо-логичког електронског кола). На тај начин остварује се побољшање укупних перформанси конкретне архитектуре. Следећа листа садржи сумарне карактеристике RISC CPU:

1. RISC CPU је дизајниран помоћу хардверског управљања са мало или нимало микро кода (програми који описују како се извршава нека инструкција на нивоу хардвера унутар микроконтролера). Имајте на уму да формати инструкција променљиве дужине обично захтевају дизајнирање микрокода (микропрограма). Већина RISC инструкција има фиксне формате, тако да дизајнирање микрокода није потребно.
2. RISC CPU извршава већину инструкција у једном циклусу.
3. Сет инструкција RISC CPU типично обухвата само регистарске инструкције, инструкције за пуњење (LOAD) и инструкције за складиштење (WRITE). Све инструкције које укључују аритметичке операције користе регистре, а операције пуњења и складиштења се користе за приступ меморији.

4. Инструкције имају једноставан фиксни формат са неколико модова адресирања.

RISC процесори су погодни као рачунари за уградњу (*embeded applications*). Уграђени контролери су део „хост“ система (основни уређај са специфичном наменом). То значи да су присуство и рад ових контролера у основи скривени унутар „хост“ система.

RISC процесори су погодни за апликације као што су обрада слике, роботика и инструментација. Кључне карактеристике RISC CPU-а које их чине идеалним за ове апликације су релативно ниски ниво интеграције у чипу и инструкције прилагођене проточној обради. Ове карактеристике омогућавају малу потрошњу енергије и брзу реализацију инструкција.

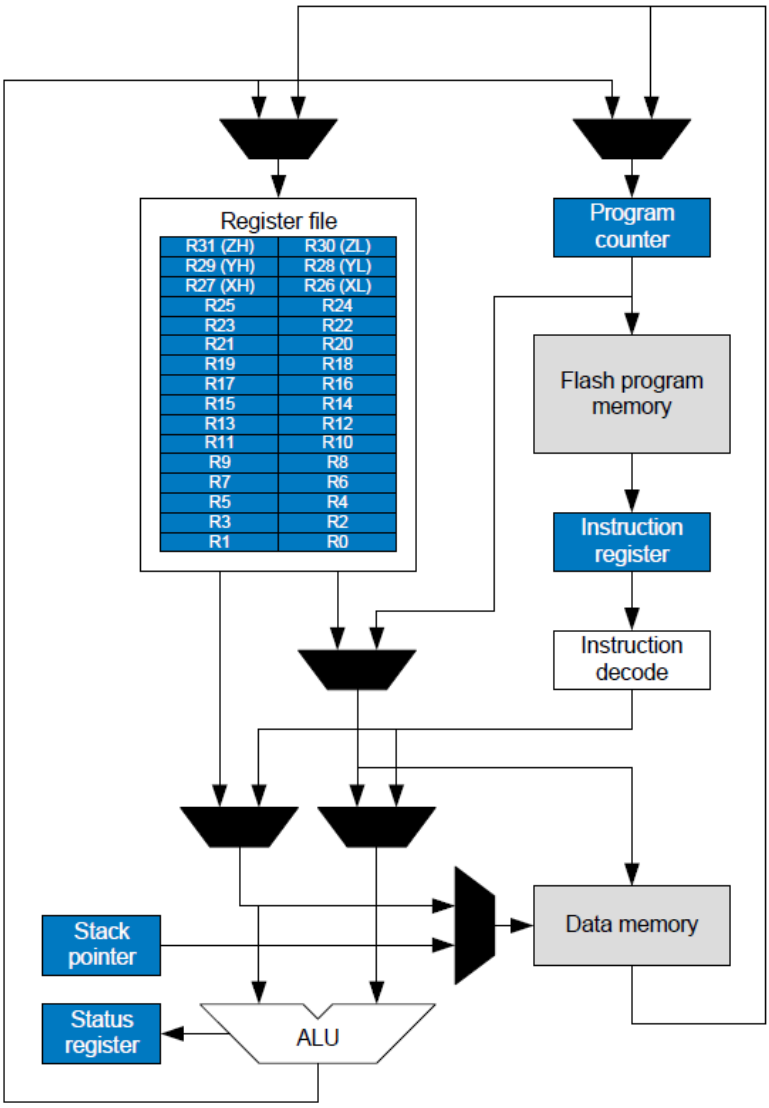
CISC процесори садрже велики број инструкција и много начина адресирања. Насупрот томе, RISC процесори садрже једноставан скуп инструкција са неколико начина адресирања. Скоро сва израчунавања се могу остварити извршењем неколико једноставних операција. RISC у основи подржава мали скуп уобичајено коришћених инструкција које се извршавају великом брзином у поређењу са CISC архитектуром са обиљем инструкција (од којих се неке ретко користе) и извршавају се далеко спорије.

У CISC-у, већина инструкција може приступити меморији док RISC садржи углавном инструкције за учитавање/складиштење. Сложени скуп инструкција CISC-а захтева комплексну контролну јединицу, чиме се захтева микропрограмирана имплементација. RISC користи хардверско управљање које је брже. CISC је теже прилагодити процесу проточне обраде у поређењу са RISC архитектуром. Предност CISC-а у о дносу на RISC јесте да сложени програми захтевају мање инструкција у CISC-у са мање циклуса њиховог дохватања, док RISC захтева велики

број инструкција за остваривање истог задатка са неколико циклуса за дохват. Међутим, RISC може знатно побољшати своје перформансе са бржим тактом, ефикаснијом паралелном обрадом (пајплајн) и оптимизацијом компајлера.

2.5. Функционални приказ типичног RISC Микроконтролера – ATmega328/P

Слика 2.8 приказује функционални блок дијаграм микроконтролера ATmega328/P. Овај микроконтролер се заснива на AVR® побољшаној RISC архитектури и биће кориштен у примерима који се обрађују у овој књизи. Због тога је у овом делу описана његова основна архитектура. Главна функција његове CPU је да обезбеди извршење програма. CPU због тога мора бити у стању да приступи меморијама, обави захтевана израчунавања, управља периферним јединицама и подржи прекиде.



Слика 2.8 Блок дијаграм AVR архитектуре

Да би максимизирали перформансе и паралелизам, AVR користи Харвард архитектуру - са одвојеним меморијама и магистралама за инструкције и податке. Инструкције које су смештене у програмској меморији извршавају се током једног основног машинског циклуса. Ово практично значи да у време док се једна инструкција извршава, следећа инструкција се

унапред преузима из меморије програма и припрема за извршење у наредном кораку, односно машинском циклусу. Овај концепт омогућава да се инструкције извршавају у сваком машинском циклусу. Програмска меморија је конципирана као репрограмабилна флеш меморија уграђена у систем, односно микроконтролер.

У овој архитектури треба уочити постојање 32 осмобитна радна регистра опште намене (регистарски фајл) са временом приступа које одговара једном радном циклусу. Ово омогућава извршење свих операција унутар аритметичко-логичке јединице (ALU) у времену од једног радног циклуса. При типичној ALU операцији, из регистарског фајла се прибављају два операнда, операција се извршава, а резултат се смешта у неки од регистара регистарског фајла у току једног радног циклуса.

Аритметичко логичка једниница подржава операције између регистара, између константи и регистара као и унарне операције. Након неке аритметичке операције, Статусни регистар (*Status register*) садржи информацију о карактеру резултата операције (на пример, да ли има прекорачења и слично).

У случају да програм мења стандардни ток због промена које су настале у хардверском окружењу микроконтролера (на пример активирање прекидача за RESET или појава спољашњег сигнала на неком од улаза микроконтролера), одговарајућа адреса програмског бројача (*Program Counter*), чува се унутар меморијског простора са посебном наменом познатог под називом показивач стека (*Stack Pointer*).

Већина AVR инструкција има формат једне шеснестобитне речи. На свакој адреси програмске меморије може се наћи шеснестобитна или тридесетдвобитна инструкција.

2.6. Основе програмских језика

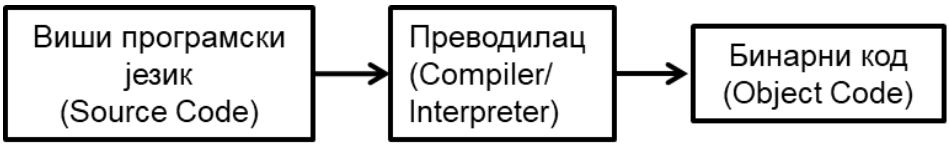
Микроконтролери се обично програмирају користећи инструкције у виду акронима изведених из њихових назива на енглеском језику (асемблерски језик). Поред асемблера, микроконтролери користе и језик високог нивоа (*high-level language*) који је далеко разумљивији за човека. Без обзира која врста језика се користи за креирање програма, микроконтролери разумеју само бинарне бројеве.

Према томе, сви програми морају бити преведени у њихове одговарајуће бинарне форме. Програмски језици се типично могу поделити на три главна типа: машински језик, асемблер и језике вишег нивоа. Програм који је написан на машинском језику састоји се од бинарних или хексадецималних операционих-кодова. Програмирање микроконтролера на овај начин је релативно тешко, јер се при писању програма користе само бројеви. Архитектура процесора унутар микроконтролера у потпуности одређује скуп инструкција које се користе у конкретном програму (скуп инструкција микроконтролера).

Програми написани на асемблеру и на језицима вишег нивоа су представљени инструкцијама чији су називи изведени из енглеског језика. Због тога је програмирање на асемблеру и језицима вишег нивоа много једноставније за програмера него на машинском језику. Међутим, при томе је неопходно да се користи одговарајући преводилац за претварање таквих програма у бинарни језик како би микроконтролер могао да изврши конкретан програм. Ово је приказано на слици 2.9.

Асемблер преводи програм написан на асемблерском језику у машински програм. Преводилац, с друге стране, претвара програм писан на језику вишег нивоа, као што је на пример C, у машински програм. Асемблерски програми као и

програми писани на језицима вишег нивоа се називају изворни кодови (*source codes*).



Слика 2.9 Превођење језика вишег нивоа у бинарни машински језик.

Програми на машинском језику су познати као објектни кодови (*object codes*). Преводилац претвара изворне кодове у објектне кодове. У тексту који следи детаљније се разматрају три основна типа програмских језика.

2.6.1. Машински језик

Микроконтролер има јединствени скуп инструкција машинског језика које је дефинисао његов произвођач. Два микроконтролера који су дизајнирани и произведени од стране различитих произвођача немају исти скуп инструкција машинског језика. Због тога програм који је написан на машинском језику за један микроконтролер неће радити на микроконтролеру другог произвођача. На најелементарнијем нивоу, програм микроконтролера може да се напише користећи инструкције у бинарном облику, односно у виду низа линија које садрже јединице и нуле:

0000111000000010

0000111100000011

0110111001000000

1110111100000011

Очигледно, овакав програм је веома тешко разумети, осим ако је програмер у стању да упамти све кодове инструкција, што је непрактично. С обзиром да је, за човека, веома неудобно да пише програме користећи само цифре 1 и 0, скоро је немогуће написати програм без грешке у првом

покушају. Такође, врло је напорно да програмер унесе овакав програм у меморију микроконтролера. Чињеница да је за овакво уношења програма неопходно постојање бројних бинарних прекидача сама по себи указује на велику вероватноћу грешке при уносу неопходног кода.

Да би се повећала ефикасност програмера при писању програма машинског кода, користе се хексадецимални, а не бинарни бројеви. Претходна програмска секвенца написана у бинарном облику, у том случају има форму:

0E02

0F03

6E40

EF03

Лакше је открити грешку у програму који је писан у хексадецималном коду, јер сваки бајт садржи само две хексадецималне цифре. У том случају, унос програма се остварује преко хексадецималне тастатуре. При томе одговарајући програм, смештен у ROM меморији, обезбеђује повезивање хексадецималне тастатуре са микроконтролером. Овај програм мора да претвори сваку инструкцију у бинарни језик машине како би микроконтролер разумео програм. Међутим, програмирање у хексадецималном коду се обично не користи.

2.6.2. Асемблерски језик

Следећи програмски ниво користи асемблерски језик. Свака линија у асемблерском програму обухвата четири поља:

- лабелу или ознаку линије
- поље мнемоника или операционог кода
- поље операнда/операнада и
- поље за коментаре

У тексту који следи приказан је пример програмских линија које су написане у асемблерском коду за микроконтролер MC68HC11.

Лабела	*Мнемоник*	*Операнд*	*Коментар*
START	LDAA	#$D6	Напуни акумулатор А подацима
	STAA	$0200	Смести податак на адресу $0200
KRAJ	WAI		Заустави извршење

Очигледно, програмирање на асемблерском језику је много згодније од програмирања на машинском језику, јер сваки мнемоник даје идеју о врсти операције која треба да се обави (на пример LDAA - *Load to accumulator A*). Стога, програмер не мора да тражи, у одговарајућој табели, нумерички облик за сваку инструкцију па је поступак програмирања значајно бржи и ефикаснији у односу на програмирање у машинском језику.

2.6.3. Језик вишег нивоа

Као што је раније речено, ефикасност програмирања се значајно повећава применом асемблерског језика у поређењу са поступком програмирања на машинским језиком. Међутим, програмер мора добро да познаје архитектуру CPU-а и конкретан скуп инструкција. Надаље, програмер мора да обезбеди операциони код за сваку операцију коју CPU мора да обави током извршења програма. На пример, за сабирање два броја, неопходно је да CPU упише први број у регистар, затим дода други број у регистар и коначно сачува резултат у меморији. Међутим, исписивање свих ових корака може бити јако заморно у случају великих програма. Поред тога,

потребно је имати пуно искуства да се постане добар програмер на асемблерском језику.

Програмски језици високог нивоа који се састоје од израза изведених из енглеског језика превазилазе све ове недостатке програмирања на машинском и асемблерском језику. Програмер не мора да буде упознат са унутрашњом структуром микроконтролера и његовом инструкцијским скупом. Такође, свака инструкција на језику вишег нивоа одговара одређеном броју инструкција у машинском језику. На пример, размотримо израз "f = a + b", за сабирање целих бројева, написан на језику вишег нивоа који се зове C. Ова наредба захтева да се саберу садржаји меморијских локација "a" и "b" и резултат сачува у меморијској локацији f. Ово је еквивалентно већем броју корака, односно инструкција на машинском и асемблерском језику.

Програмски језик C је врло популаран и најчешће се користи за програмирање микроконтролера. Језик на вишем нивоу је проблемски оријентисан језик. Програмер не мора да познаје детаље о архитектури микроконтролера и његовом скупу инструкција. У основи, програмер следи правила одређеног језика који се користе за решавање проблема. Друга предност је што програм који је написан на одређеном језику на високом нивоу може да се изврши на два различита микроконтролера, под условом да оба разумеју тај језик. Наравно, при томе, оба морају поседовати сопствени компајлер за превод програма са C језика на свој машински језик. Најчешће су једино неопходне мале модификације програма за дефинисање улазно излазних операција у складу са каркатеристикама конкретног микроконтролера.

Микроконтролери су често подржани са одговарајућим програмом познатим под називом преводилац (*interpreter*). Овај програм постоји као део пакета за развој софтвера.

Интерпртер „исчитава“ сваку наредбу која је написана у вишем програмском језику, на пример F=A+B, и усмерава микроконтролер да реализује операције потребне за њено извршење. Другим речима, интерпретер претвара сваку наредбу у наредбе машинског језика, али не и цео програм пре извршења. Дакле, он не генерише објектни програм. Значи, извршење програма се остварује кроз процес узастопног превођења и извршења сваке наредбе, односно линије програмског кода који је написан у вишем програмском језику.

Компајлер, међутим, претвара све наредбе неког програма у скуп наредби машинсог језика и генерише објектни програм који се чува у меморији. Овај објектни програм се затим извршава од стране CPU-а како би се реализовао задатак који је дефинисан у програму вишег нивоа.

Укратко, интерпретер омогућава да се свака наредба изврши, након њеног очитавања, док компајлер претвара комплетан програм вишег нивоа у објектни програм који чува у меморији, а затим се исти активира од стране корисника ради извршења.

2.7. Одабир програмског језика

Савремени C компајлери за микроконтролере генеришу веома ефикасне кодове. С обзиром да је C језик вишег нивоа и садржи улазно-излазне наредбе веома је популаран међу програмерима па се често користи за програмирање микроконтролера. Програмирање у асемблеру, с друге стране, игра важну улогу у разумевању унутрашње архитектуре микроконтролера и може бити корисно при откривању грешака у програмима.

2.8. Сажетак

Микроконтролери садрже централну процесорску јединицу, меморију тајмере, AD и DA претвараче, аналогне компараторе, серијске улазно излазне јединице и друге периферне уређаје. Сви уређаји микроконтролера повезани су путем такозване системске магистрале која садржи адресну магистралу, магистралу података и управљачку магистралу.

На адресној магистрали пренос информација се одвија у само једном правцу, од CPU-а ка меморији или I/O јединици.

На магистрали података, подаци могу да струје у оба смера, односно ка или од CPU-а. Због тога је ово двосмерна магистрала.

Контролна магистрала се састоји од више сигнала који се користе за синхронизацију рада појединих микрорачунарских елемената. CPU шаље неке од ових контролних сигнала другим елементима како би указао на врсту извршене операције.

За дизајнирање микроконтролера се користе две врсте CPU архитектура, фон Нојман (Princeton) и Харвард.

У архитектури фон Нојман користи се иста адресна магистрала и магистрала података било да се приступа програмима или подацима. То значи да се програмима (инструкцијама) и подацима не може приступити истовремено.

Харвард архитектура микроконтролера користи засебне јединице за смештај програма и чување података упоредо са засебним магистралама за инструкције и податке па ови микроконтролери могу истовремено извршавати инструкције и приступати подацима.

Конвенционални процесори извршавају програме као низ инструкција. Модерније верзије микроконтролера користе

поступак познат под називом проточна обрада. На овај начи обезбеђује се повећање пропусне моћи система.

У погледу инструкцијског скупа уочавају се два типа CPU архитектуре: RISC (смањени рачунски сет инструкција) и CISC (комплексни компјутерски сет инструкција).

CISC процесори садрже велики број инструкција и много модова адресирања. Насупрот томе, RISC процесори садрже једноставан скуп инструкција са неколико начина адресирања. Скоро сва израчунавања се могу остварити извршењем неколико једноставних операција. RISC у основи подржава мали скуп уобичајено коришћених инструкција које се извршавају великом брзином у поређењу са CISC архитектуром са обиљем инструкција (од којих се неке ретко користе) и извршавају се далеко спорије. Циклус дохвата и извршења (fetch/execute) у случају CISC архитектуре одвија се типично применом споријег такт сигнала.

У CISC-у, већина инструкција може приступити меморији док RISC садржи углавном инструкције за учитавање/складиштење. Сложени скуп инструкција CISC-а захтева комплексну контролну јединицу, чиме се захтева микропрограмирана имплементација. RISC користи хардверско управљање које је брже. CISC је теже прилагодити процесу проточне обраде у поређењу са RISC архитектуром. Предност CISC-а у о дносу на RISC јесте да сложени програми захтевају мање инструкција у CISC-у са мање циклуса њиховог дохватања, док RISC захтева велики број инструкција за остваривање истог задатка са неколико циклуса за дохват. Међутим, RISC може знатно побољшати своје перформансе са бржим тактом, ефикаснијом проточном обрадом (пајплајн) и оптимизацијом компајлера.

Микроконтролери се обично програмирају користећи инструкције у виду акронима изведених из њихових назива на енглеском језику (асемблерски језик). Поред асемблера,

микроконтролери користе и језик високог нивоа који је далеко разумљивији за човека.

Сви програми морају бити преведени у њихове одговарајуће бинарне форме. Програмски језици се могу поделити на три главна типа: машински језик, асемблер и језике вишег нивоа. Програм који је написан на машинском језику састоји се од бинарних или хексадецималних операционих-кодова. Програмирање микроконтролера на овај начин је релативно тешко, јер се при писању програма користе само бројеви.

Програми написани на асемблеру и на језицима вишег нивоа су представљени инструкцијама чији су називи изведени из енглеског језика па је, у том случају, програмирање много једноставније за програмера него на машинском језику. При томе је неопходно да се користи одговарајући преводилац како би се такви програми превели у бинарни језик који је разумљив за микроконтролер.

Асемблер преводи програм написан на асемблерском језику у машински програм. Компајлер претвара сваку наредбу у скуп наредби машинсог језика и генерише објектни програм који се чува у меморији.

Асемблерски програми као и програми писани на језицима вишег нивоа се називају изворни кодови (source codes).

Питања

1. Наведите основне блокове микроконтролера.
2. Објасните структуру и улогу системске магистрале.
3. Објасните разлику између операција WRITE и READ.
4. Наведите основну разлику између фон Нојман и Harvard архитектуре.
5. Објасните улогу системског сата у оквиру микроконтролера.
6. Објасните суштину пајплајн реализације инструкција.
7. Упоредите RISC и CISC архитектуру.
8. Објасните разлику између машинског језика, асемблера и програмских језика вишег нивоа.

3

ARDUINO C

Поглавље започиње описом градивних блокова који се појављују без обзира на врсту програмских језика. Дефинишу се основни програмски кораци које треба имати у виду при развоју програмске подршке за решење неког конкретног проблема. На крају је дат приказ програмског окружења у којем се развија програм за системе типа ARDUINO.

Програмски језик који користи ARDUINO окружење није стандардна верзија C језика. Могло би се рећи да је ово заправо прилагођена верзија језика која се односи као подскуп у односу на стандардну верзију C језика. Ово је сасвим лако разумети ако се има у виду да прилагођена верзија коју ћемо надаље звати ARDUINO C нема поједине стандардне карактеристике. Међутим, одсуство тих карактеристика није никакав проблем јер се недостајуће особине могу лако надокнадити применом решења која, наравно, понекад нису тако елегантна као у случају стандардне верзије C језика.

3.1. Градивни блокови свих програмских језика

Сви програмски језици су сачињени од четири основна елемента:

1. Израза
2. Наредби
3. Блокова наредби и
4. Функцијских блокова

Последњи елемент, функцијски блокови, могу се назвати различитим именима на различитим језицима, као што су "Процедуре", "Потпрограми"; или неким другим именом на мање познатим језицима. Без обзира на њихово име, функцијски блокови су делови програмског кода који су дизајнирани да реше неки уско дефинисан задатак. Комбиновањем ових елемената на специфичан начин решава се конкретан проблем.

3.1.1. Изрази

Израз се креира комбиновањем операнада и оператора. Једноставно речено, операнд је неки од података на који оператор делује. Оператор је математичка или логичка операција која се обавља над једним или више операнада. На пример:

a + b

m – 3000

g < d

су примери израза. У првом примеру сабирају се операнди a и b (оператор +). У другом примеру нумеричка константа 3000 (операнд) се одузима (оператор -) од операнда под именом m. У последњем примеру, операнд g се упоређује са операндом d да би се утврдило да ли је g мањи од (оператор <) d. У овом посљедњем примеру, уместо математичког оператора, користи се релацијски оператор (тј. оператор "мање од" или <). У сва три примера, два операнда се користе заједно са једним оператором како би се формирао израз. Први пример је израз сабирања, други је израз за одузимање, а последњи пример представља релацијски израз.

У сваком од ових израза постоје два операнда и један оператор. Због тога се каже да су ово изрази који користе бинарни оператор. Бинарни оператори (нпр. +, - и <) увек користе два операнда. Још једна важна ствар коју треба имати у виду јесте да ће сваки израз на крају произвести резултат у облику конкретне вредности. (Постоје и унарни оператери који имају само један операнд и тројни оператори који захтевају три операнда). Међутим, бинарни оператори су најчешћи у C-у.

Изрази се могу комбиновати. На пример, претпоставимо A = 1, B = 2 и C = 3. Можете написати сложени израз као:

A + B + C

С обзиром да сви изрази имају вредност као решење, може се решити први израз A + B, тако да се претходни израз поједностави:

1 + 2 + C.

Пошто први израз сада чине чисти бројеви, прво се израчунава његова вредност 3. Затим се решава израз:

3 + C

Уочимо шта се овде догодило. Првобитно је постојао сложени израз са два оператора и три операнда. Затим је решен један од израза (тј. A + B) до 3. Међутим, током овог процеса, сложени израз је сведен на један израз, 3 + C. Коначно се решава преостали израз:

3 + C

3 + 3

6

тако што је сложени израз са два подизраза сведен на једну вредност, 6. Често ћете чути да се овај процес поједностављивања комплексног израза назива факторизација израза.

Шта се дешава у случају релацијског израза који је представљен описом g<d? Претпоставимо да је g = 5 и d = 4. Тада следи:

g < d

5 < 4

false

У резултату овог релацијског израза добија се false (нетачно) јер 5 није мање од 4.

Приметимо да смо у решењу добили нешто што није бројна већ логичка вредност. Ово је коректно јер већина програмских језика третира решења релацијских израза као вредности (1 за true, односно тачно и 0 за false, односно нетачно).

3.1.2. Наредбе

Наредба је комплетна C инструкција за рачунар. Све наредбе у C-у се завршавају са тачком и запетом (;). Следећи примери су наредбе у C-у:

```
i = 50;

a = b + c;

m = d / 2;
```

У првом примеру, знак једнакости (=) се назива оператором додељивања и користи се за "доделу" вредности, са десне стране знака једнакости, променљивој на левој страни оператора доделе. Дакле, у првој наредби, вредност 50 је додељена променљивој i. Овај пример није ништа друго до израз дефинисан помоћу оператора доделе са тачком и запетом на крају линије. Операнди су 50 и променљива i.

Шта је променљива? Једноставно речено, променљива није ништа друго до локација у меморији којој је додељено име. У поглављу које следи биће више речи о променљивама.

У другој наредби, појављује се сложени израз са тачком и запетом на крају. У овом примеру вредност која треба да се додели променљивој a још није позната, тако да је неопходно прво решити израз b + c да би се добила вредност. Ако су b = 4 и c = 5, онда се добија решење сложеног израза:

```
a = b + c

a = 4 + 5

a = 9
```

Последњи израз додељује вредност 9 променљивој a. Додавањем тачке и запете на крају линије, израз постаје наредба којом се захтева да променљива a промени своју вредност на 9. Подсетимо се да компајлер преводи изворну верзију програма у низ јединица и нула које микроконтролер

разуме. Тачка и запета на крају реда управо сигнализирају компајлеру где је крај текста који одговара конкретној наредби. Уколико имате сложену наредбу као:

```
x = a + b – c + g + h + k;
```

тада компајлер мора да разреши све основне изразе (a + b, c + g, h + k) пре него што утврди коју нову вредност треба доделити променљивој x. Тачка и запета на крају наребе управо указују компајлеру да има све међуинформације које су потребне да се изврши конкретна наредба. Изостављање тачке и запете на крају наредбе представља најчешћу почетничку грешку при писању C програма.

3.1.3. Приоритет оператора

Претпоставимо да имате наредбу састављену од неколико израза:

```
j = 5 + k * 2;
```

где је k= 3 а звездица (*) означава оператор множења. Поставља се питање да ли је j= 16 (тј., 8x2) или је j= 11 (тј., 5+6=11). Неизвесност у погледу резултата следи из непознавања приоритета, односно редоследа којим операције треба да се изврше. Који од следећих поступака даје тачан резултат:

```
j = 5 + k * 2;          j = 5 + k * 2;
j = 5 + 3 * 2;          j = 5 + 3 * 2;
j = 8 * 2;              j = 5 + 6;
j = 16;                 j = 11;
```

Очигледно, резултати се разликују због редоследа којим решавамо сложени израз. C решава такве нејасноће додељивањем одговарајућег нивоа приоритета сваком операторту. Приоритет оператора се односи на редослед

операција при решавању сложеног израза. Делимична табела приоритета може се видети у Табели 3-1.

Табела 3-1

Приоритет	Оператор
1	*, /, %
2	+, -

У табели 3-1, можете видети да операције множења (*), дељења (/) и оператор који одређује остатак после дељења (%) имају виши приоритет у односу на сабирање и одузимање. Због тога, у горе наведеним изразима, тачан одговор за j је 11, јер се израз множења решава пре израза сабирања. Уколико су приоритети сукцесивних оператера исти, онда се решење добија извршавањем математиких операија с лева на десно.

Решавање подизраза на овај начин указује да су математички оператори лево асоцијативни. Термин лево асоцијативни значи да се, у случају оператора са једнаким приортетима, решавање подизраза обавља у редоследу с лева на десно. Будући да има више оператора него што је приказано у Табели 3-1, проширићемо таблицу приоритета док сазнамо нешто више о C-у.

3.1.4. Блокови наредби

Блок наредби се састоји од једне или више наредби које су груписане тако да их преводиоци виде као да су појединачни изрази. Блок наредби почиње са знаком { и завршава са закључним знаком }. Другим речима смешта се између витичастих заграда. Све наредбе између заграда формирају тело блока наредби. Унутар блока наредби можете ставити било коју врсту наредбе. У поглављима која следе

видећете пуно примера, а за сада, само замислите блок наредбу као део софтвера који је дефинисан садржајем између отворене и затворене витичасте заграде.

3.1.5. Функцијски блокови

Функцијски блок је програмска целина која је дизајнирана да изврши један задатак. Овај блок такође чине наредбе скоје су смештене између витичастих заграда. Међутим, функцијски блокови се могу замислити као "црне кутије" у којима су скривени детаљи о томе како се нешто ради.

Већ је речено да сваки програм можете да замислите као творевину сачињену од четири основна дела који се описују у овом одељку. Заправо, остатак ове књиге није ништа друго него приказ поступака како да користите ове једноставне делове на ефикасан начин да решите одређени програмски проблем. Суштина проблема крије се управо у чињеници да постоји бесконачан број комбинација ових елемената како би се формила програмска целина. При томе ће неке од ових комбинација дати жељени резултат а друге неће уопште моћи да раде. Заправо, чак и ако сачините програм који решава конкретни проблем, не значи да не постоји ефикаснији начина за остваривање истог задатка. На пример, претпоставите да је потребно сортирати групу бројева у виду листе, од најмањег до највећег у групи. Постоји више начина сортирања листе бројева у растућем редоследу, свака са својим предностима и мањкавостима. У ствари, видећете да се опсег ваших избора повећава како сазнајете више о програмирању у општем смислу. Што више искуства и знања о програмирању стекнете, више ћете бити у стању да направите елегантно решење за програмски проблем. При томе треба имати на уму да се повећањем сложености проблема повећава и број могућих решења конкретног проблема.

3.2. Програмски кораци

Сваки програм можете свести на пет основних програмских елемената или корака. Када први пут започнете дизајнирање програма, требало би да размислите о том програму у смислу следећих пет програмских корака.

3.2.1. Иницијализација (Initialization step)

Сврха корака иницијализације је успостављање окружења у којем ће се програм покренути. На пример, ако сте икада користили EXEL, Word или сличне програме, наредба File често садржи листу недавно коришћених датотека. Интернет претраживачи вам омогућавају да дефинишете почетну страницу. Програм за штампање често има подразумевани штампач. Програм базе података често успоставља подразумевану мрежну везу. У свим овим случајевима, подаци се дохватају од некуд (тј. датотеке са подацима, меморије, ЕЕПРОМ, регистра) и користе за успостављање неког основног окружења у којем се програм покреће.

3.2.2. Дефинисање улаза (Input step)

Готово сваки рачунарски програм има задатак који је дизајниран да преузме неке постојеће информације, обради их на неки начин и прикаже ново стање тих информација. Заправо, можемо рећи да је дефинисање улаза, односно улазни корак, секвенца програмских наредби (инструкција) које су неопходне за стицање информација потребних за решавање конкретног задатка.

3.2.3. Обрада података (Process step)

Централни део сваког програма садржи скуп инструкција које су креиране да преузму улазне податке и наставе њихову обраду како би се добио нови скуп података.

Наравно, сам поступак обраде се дефинише у зависности од конкретног задатка.

3.2.4. Дистрибуција резултата обраде (Output step)

Након што се заврши корак обраде података, новодобијени резултати се обично приказују на неком показивачу, али могу послужити и као сигнали за управљање неким процесом или се преносе у неки други програм. Очигледно, дистрибуција резултата обраде, односно излазни корак треба да обезбеди кориштење резултата обраде података.

3.2.5. Завршетак програма (Termination step)

Завршни корак је одговоран за "чишћење" након што програм обави свој задатак. У десктоп апликацијама, уобичајено је да се током овог корака обаве иницијалне радње у супротном смеру. То значи да уколико програм чува траг о недавно кориштеним датотекама, онда се у овом кораку мора ажурирати та листа. Исто тако ако је током иницијализације успостављена веза са базом података или штампачем, онда завршни корак треба да обезбеди затварање те везе, тако да су неискориштени ресурси доступни систему.

Међутим, када говоримо о микроконтролерима, треба имати у виду да они најчешће обављају неки задатак кроз циклично надзирање стања извора информација (нпр. стање сензора, промене у регистрима за пријем и слање података) како би подржали непрекидно извршење конкретног задатка. Наравно, то не значи да овај корак нема смисла када говоримо о програмима који се пишу за микроконтролере. Зависно од задатка који обављају могу настати ситуације када је неопходно предузимање ванредних активности у смислу сервисирања система у којем микроконтролер обавља своју функцију. Део програмске подршке може да буде дизајниран

тако да се упоредо са основним задатком надзире и исправност комуникационих линија или самих сензора. Уколико се уочи било какво одступање од очекиваног рада микроконтролер треба да генерише одговарајуће акције као што су на пример сигнализација да нешто не ради како треба, могући тип грешке, меморише тренутно стање система и слично како би се правовремено отклонио уочени недостатак.

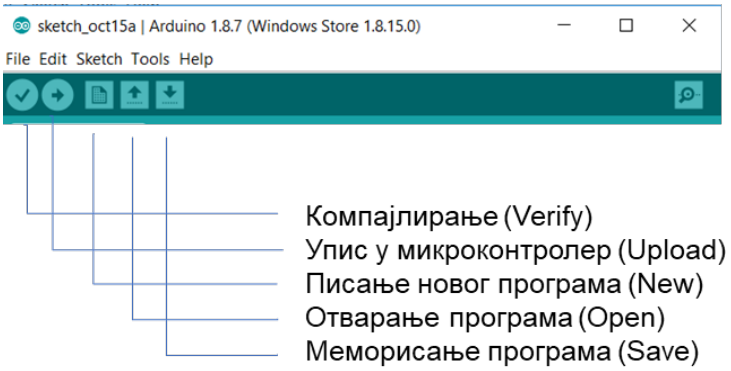
Која је сврха ових пет програмских корака? Они треба да служе као полазна тачка за дизајнирање програма. Чак и једна до две реченице за сваки од корака вероватно је довољна да бисте почели са дизајном и кодирањем датог програма. Алгоритам није ништа друго до формално упутство како да се поступа са датим скупом улазних података у циљу добијања неопходног резултата. Алгоритам је као рецепт или скуп нацрта: Они описују шта треба да урадите да бисте постигли жељени циљ или крајњу тачку. Тако је и са програмирањем: Пет програмских корака се могу користити за формулисање плана који указује како да се реши задани проблем програмирања. Иако су алгоритми блиско повезани са корацима 2 и 3 (тј. улаз и обрада), пет програмских корака треба да вам помогну да формулишете алгоритам за решавање било ког задатка.

3.3. Програмско окружење за развој програма

Постојећа програмска подршка нуди могућнст за развој конкретног програма као и његово смештање у меморију микроконтролера након чега је исти спреман за рад у конкретном окружењу.

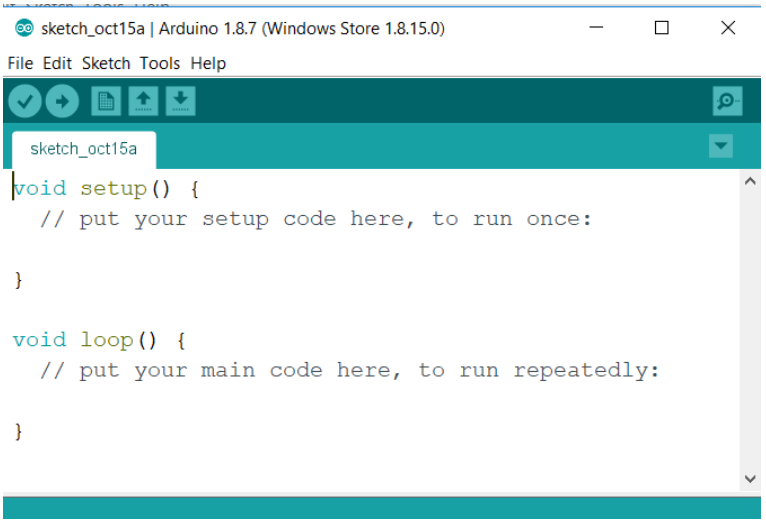
Након што покренете конкретно развојно окружење на екрану ће се појавити прозор (слика 3.1) који садржи одговарајуће падајуће менije и неколико иконица за комфорнији рад. Детаљи падајућих менија се лако упознају јер као и обично садрже сугестивне називе, а ту је и помоћ

(Help) обезбеђена у оквиру добро креираног садржаја који се ослања на интернет у случају да су вам потребне додатне инфорамције.



Слика 3.1 Приказ основног менија за развој прогрма у ARDUINO C програмском окружењу

Писање новог програма започиње активирањем иконице у средини реда (симбол који већина програмске подршке користи да означи текстуални документ). Након отварања појавиће се екран приказан на слици 3.2.



Слика 3.2 Садржај екрана при иницијализацији писања новог програма

У конкретном случају аутоматски је генерисан натпис sketch_oct15a што указује да се ради о програму који је генерисан 15. октобра. Наравно, ви можете сачувати програм под именом које ћете сами дефинисати уколико уђете у мени File (опција Save as). Уколико вам предложени наслов не смета једноставно активирајте крајњу десну иконицу (означена стрелицом на доле) и ваш програм ће бити сачуван у меморији рачунара за каснији рад.

Након писања програма следи верификација која подразумева компајлирање и покреће се активирањем прве иконице с леве стране (означена знаком за верикфикацију). Уколико је све у реду (нема порука о постојању грешака) активираћете суседну иконицу (означена стрелицом која је усмерена у десно) како бисте програм сместили у меморију конкретног микроконтролера. Након тога миконтролер је спреман за рад у конретном окружењу, односно за извршење додељеног задатка.

Уколико желите да отворите програм који већ постоји у одговарајућем директоријуму рачунара активирајте иконицу са стрелицом која је усмерена на горе. Отвориће се одговарајући прозор где можете пронаћи жељени програм.

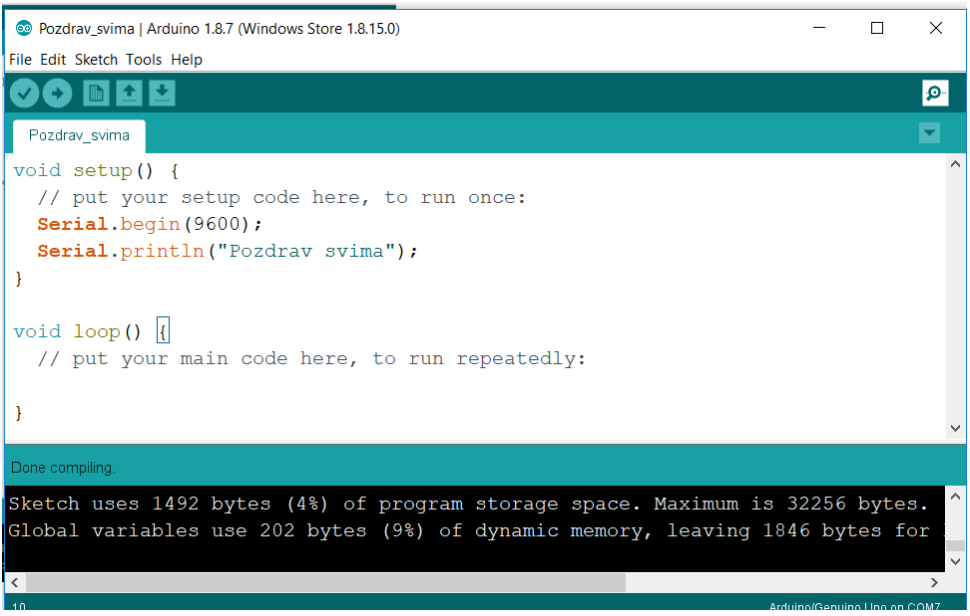
Приметимо да нови програм садржи два функцијска блока (void setup и void loop):

```
void setup() {
        // put your setup code here, to run once:
}
void loop() {
        // put your main code here, to run repeatedly:
}
```

У првом од ових блокова наћи ће се све наредбе које је неопходно да се изврше само једном, односно при

активирању програма (након укључења микроконтролера или његовог „ресетовања“). Уочимо да су одговарајући коментари назначени навођењем две косе црте, без размака, на почетку одговарајућег реда. Нагласимо да коментари не заузимају меморијски простор микроконтролера. Косе црте управо указују компајлеру да су ово коментари и биће изостављени у поступку превођења програма у машински код. У случају да коментари заузимају простор који је већи од једног реда морају започети знацима /*, а завршити са знацима */.

У тексту који следи приказан је кратак програм (Pozdrav_svima) који омогућава испис кратке поруке на екрану рачунара. Наравно, за ово је потребно да микроконтролер остане повезан са рачунаром. Користећи монитар који је повезан преко серијске везе са мокроконтролером (активира се уласком у мени Tools, опција Serial monitor) могу се прочитати поруке које су раније дефинисане у овом програму.



Слика 3.3 Програм Pozdrav_svima

Приметимо да програм садржи три једноставне наредбе. Прва од њих (Serial.begin(9600);) заправо представља корак иницијализације јер се заснива на примени функцијског потпрограма који обезбеђује серијски пренос ка монитору. Рецимо, на овом месту, не улазећи у детаље, да наведени параметар (9600) дефинише брзину преноса. О овоме ће бити више речи у неком од наредних поглавља.

Да би се омогућио испис поруке неопходно је позвати[1] потрпрограм са називом (Serial.println). Порука која треба да се испише мора бити наведена између знакова навода. У програму је уведена и наредба (Serial.println("\"M I K R O R A C U N A R I\" ");) која приказује како да се знакови навода појаве као део поруке. Тада је неопходне да се испред њих напише коса црта у назад. Уколико се ова црта изостави при комплајирању ће бити дојављена порука о постојању грешке. У наставку текста приказан је нешто сложенији програм (BLINK) који омогућава аутоматско укључивање и искључење светлеће диоде (Light emiting Diode - LED). Овај програм ћете наћи међу примерима (Мени File, опција Examples>01.Basics>Blink) који се појављују у оквиру развојног софтвера. Приметимо да се на почетку програма појављује вишелинијски коментар који је због тога смештен између знакова /* и */.

```
/*  Blink
Turns an LED on for one second, then off for one second, repeatedly.
This example code is in the public domain. */
void setup() {
        pinMode(LED_BUILTIN, OUTPUT); // initialize digital pin
LED_BUILTIN as an output.
}
```

[1] Израз који програмери често користе када се извршење програма прусмерава на наредбе унутар потпрограма.

```
void loop() {
        digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is
the voltage level)
        delay(1000);                              // wait for a second
        digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making
the voltage LOW
         delay(1000);                              // wait for a second
        }
```

Слика 3.4 Програм BLINK

Развојно хардверско окружење (ARDUINO Nano) садржи између осталог и светлећу диоду која је повезана са ножицом 13. У првом делу програма (void setup) користи се наредба

```
pinMode(LED_BUILTIN, OUTPUT);
```

којом се дефинише да је пин 13 (нове верзије софтвера имају уграђен службени назив LED_BUILTIN за овај пин) употребљен као излаз (OUTPUT).

Део програма (void loop) садржи наредбе

```
digitalWrite(LED_BUILTIN, HIGH);
digitalWrite(LED_BUILTIN, LOW);
```

којим се пин 13 доводи на виши (HIGH) и нижи (LOW) логички ниво како би се диода укључила и искључила у трајању које је дефинисано наредбом delay(1000);. Број 1000 означава време у милисекундама.

3.4. Сажетак

Програмски језик који користи ARDUINO окружење није стандардна верзија C језика већ је прилагођен у складу са карактеристикама конкретног микрококнторлера.

Сви програмски језици су сачињени од четири основна елемента: израза, наредби, блокова наредби и функцијских блокова.

Сваки програм може се свести на пет основних програмских елемената које чине: иницијализација, дефинисање улаза, обрада података, дистрибуција резултата обраде и процедуре које треба обавити непосредно на самом крају програма, односно пре завршетка његове употребе.

Питања

1. Објасните разлику између израза и наредбе.
2. Како се дефинишу блокови наредби у случају ARDUINO програма?
3. Шта су функцијски блокови?
4. Каква је улога void setup функције у ARDUINO програмима?
5. Каква је улога void loop функције у ARDUINO програмима?
6. Како се дефинишу коментари у ARDUINO програмима?
7. Како се успоставља комуникација између микроконтролера и серијског монитора у ARDUINO програмима?

4

ПОДАЦИ

У овом поглављу описане су основне карактеристике типова података, поступак дефинисања и декларисања променљивих као и конверзија података из једног типа у други.

Подаци су предмет обраде у програмима. Сваки податак има особине које одређују његову припадност неком од типова података. Тип податка одређује опсег вредности које може да узме (самим тим и заузеће меморије, односно број неопходних меморијских локација за његово смештање) као и операције које могу да се изводе над податком.

Подаци могу да се представе помоћу вредности или помоћу идентификатора. Уколико су подаци представљени помоћу вредности они представљају константе. Константе, у складу са својим именом, не могу мењати вредности током извршења програма. Наравно константе могу да се представе помоћу идентификатора. Добро позната вредност 3,14159 може да се представи идентификатором PI. Константе представљене идентификатором не заузимају простор у меморији већ представљају саставни део наредбе.

Уколико подаци, смештени у меморију рачунара, могу да мењају своју вредност током извршења програма онда кажемо да такви подаци представљају променљиве или варијабле.

Посебну врсту променљивих података представљају непостојани, подаци (*volatile data*) који могу да се промене у случајним временским тренуцима. Ово су подаци који се сређу у регистрима периферних уређаја.

Подаци могу да буду прости и сложени. Прости не могу да се расставе на мање елементе у циљу независне обраде. Због тога се каже да они немају структуру и називају се неструктуирани, односно скаларни подаци. Сложени подаци се могу раставити на простије како би се независно обрађивали. С обзиром да ови подаци имају одређену структуру називају се структурирани подаци.

Табела 4-1 садржи листу основних типова података. Сваки од типова који су приказани у овој табели представљају

кључне речи у језику ARDUINO C. Појам кључна реч односи се на било коју реч која има специјално значење за C компајлер. Због тога се ове речи не могу користити за неке друге сврхе (именовање променљивих и функцијских потпрограма). Свака погрешна употреба кључне речи биће дојављена као грешка након превођења изворног кода.

При именовању променљивих и функцијских потпрограма могуће је користити:

1. Сва слова енглеске абецеде, мала (a - z) и велика (A – Z)
2. Доњу црту (_)
3. Цифре 0 до 9, али не као први знак у имену.

Све остало није допуштено (кључне речи, знаци интерпукције, специјални невидљиви карактери). Наведимо неколико примера коректно дефинисаних имена:

Alfa	alfa	masa	struja
suma	dan1	Mesec1	_sistem

У складу са наведеним ограничењима можемо лако закључити да следећа имена нису коректна:

-Alfa	^alfa	4dan	@adresa
%suma	not-OK	a&b	kako?

Табела 4-1

Тип	Димензија (Бајт)	Опсег вредности
boolean	1	Само за логичке вредности true и false
char	1	–128 to +127
unsigned char	1	0 to 255
byte	1	0 to 255
Int	2	–32,768 to 32,767
unsigned int	2	0 to 65,535
word	2	0 to 65,535
long	4	–2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
float	4	–3.4028235E+38 to 3.4028235E+38
double	4	–3.4028235E+38 to 3.4028235E+38
string	?	('\0') обавезно на крају низа података типа карактер
String	?	Сви подаци се третирају као јединстаена целина – објекат
array	?	Низ података који су означени јединственим именом
void	0	Кључна реч која се користи како би се нагласило да функција не враћа никакву вредност након њеног позива

Како заправо дефинисати добро име? Имена треба да буду таква да јасно описују шта променљива представља или, ако је у питању потпрограм, шта конкретни потпрограм ради. При томе треба да су кратка ради лакшег писања и прегледности програма. Понекад се ово постиже избацивањем самогласника. У случају када се име састоји од више речи погодно је сваку реч започети великим словом као у примерима који следе:

myData rfDrive mtrCntrl offOutputStage

При томе не заборавите да ARDUINO C прави разлику између малих и великих слова па myData и MyData нису исте променљиве.

4.1. Бројчани типови података

Пре него што наставимо, вратимо се за тренутак на причу о бројним системима. Познато је да дигитални рачунари користе бинарни бројни систем који се заснива на примени само две цифре 1 и 0. Ово омогућава једноставну практичну реализацију аритметико логичких кола јер се захтева примена само два стања за дефинисање ове две цифре (укључено/искључено, односно води или не води струју). Информација коју чини једна бинарана цифра, у рачунарској литератури, се назива бит. При томе је за груписање ових цифара у виду осам бита усвојен назив бајт. Због тога се димензија података, односно величина меморијског простора која је потребна за њихово смештање најчешће изражава преко броја бајта, а не броја бита.

Као и у случају декадског система, тежинска вредност сваке цифре одређена је њеним положајем унутар низа цифара које чине неки бинарни број. Табела 4-2 управо даје упоредни приказ података у бинарном и декадском систему за

бројеве чија димензија не прелази садржину која одговара једном бајту.

Табела 4-2

Позиција бита	7	6	5	4	3	2	1	0
Потенција од 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Децимална вредност	128	64	32	16	8	4	2	1
Бинарни број	0	1	0	0	0	0	0	1
Децимална вредност		64						1

Видимо да позиција бита одређује потенцију броја 2, односно тежинску вредност конкретне цифре. Тако ће, на пример, јединица на позицији 6 одговарати потенцији $2^6 = 64$. Познато је да нулта потенција неког броја има вредност један па тежинска вредност јединице на позицији нула има управо ову вредност ($2^0 = 1$). С обзиром да се вредност броја добија као сума умножака његових цифара и припадајућих тежинских вредности бинарни број (01000001), у Табели 4-2, представља вредност 65 у декадском бројном систему.

Уочиимо шта се дешава уколико цифре бинарног броја померимо за једно место у лево. Свака цифра поприма два пута већу вредност. Очигледно, свако додатно померање у леву страну поново удвостручава претходну вредност. Аналогно томе свако померање за једно место у десно одговара дељењу са два. ARDUINO C има уграђену подршку за обављање операција померања бита. Њихово коришћење и значај биће детаљно описани у каснијим поглављима.

Бројчани типови података се појављују као целобројни (*integer*) и реални (*real*) тип. Целобројни (*int*) тип података омогућава појављивање бројева у облику позитивних или негативних целих вредности. Резултат дељења два целобројна податка увек је цео број (на пример 5/4, резултат је 1 а не 1,25). Опсег вредности које може имати било који

податак одређује величина меморијског простора који је предвиђен за њихово смештање. У конкретном случају (ARDUINO C) целобројни подаци се смештају унутар меморијског простора величине 1, 2 или 4 бајта. У случају неких других језика (Java, C++) целобројни подаци се смештају у меморијски простор величине 32 бита, односно 4 бајта. Опсег позитивних вредности које узима целобројни податак може се удвостручити уколико изоставимо могућност коришћења негативних вредности. Бит највише вредности који се користи за дефинисање предзнака, у овом случају, је слободан тако да располажемо са могућношћу да дефинишемо дупло већи број бинарних комбинација (неозначени целобројни подаци - *unsigned int*).

У случају потребе приказа целобројних податак у опсегу који превазилази могућности типа *int* могуће је увести дефинисање бројчаних променљивих применом типова *long* и *unsigned long*. Ови типови података смештају се у меморијски простор величине четири бајта што значајно повећава број расположивих бинарни комбинација, односно опсег података. Међутим, уколико сте сигурни да ће опсег података остати унутар граница које одговарају двобајтном формату (*int*) употребу (*long*) типа податак треба избећи јер се тиме штеди меморијски простор а и поједине операције као што је на пример померање (*shuffting*) брже се реализују на овом формату него у случају четири бајта.

Реални тип података (*float*, од назива *floating point*) подразумева могућност записа разломљених бројева у облику mEk, где се m назива мантиса, а k експонент. Велико слово E означава декадску бројну базу. Мантиса је децимални број који може да има предзнак, а целобројни и децимални део се обавезно раздвајају тачком. Експонент је цео број који може да садржи и предзнак. Поједини делови потпуног записа (облик mEk) могу и да се изоставе, али обавезно мора да

постоји тачка или E како би се реални подаци разликовали од целобројних. На пример:

-12.1E+2	($-12{,}1 \cdot 10^2 = -1210$)
6.	(6,0)
.35	(0,35)
1E5	($10^5 = 100000$)

Реални бројеви смештају се у меморијском простору величине четири бајта. У многим језицима релани бројеви могу да се запишу прецизније (у типу *double*) што захтева дупло већи меморијски простор. У случају језика ARDUINO C типови података *float* и *double* се не разликују, односно заузимају исти меморијски простор (четири бајта) па имају и исти опсег могућих вредности. Меморисјки простор величине четири бајта омогућава приказ *float* података у опсегу од приближно $\pm 3{,}4 \cdot 10^{38}$. Међутим, треба имати у виду да је највећа прецизност коју можемо очекивати, у случају језика ARDUINO C, у границама од седам децималних цифара. Ово практично значи да без обзира што број, у случају *float* података, можете описати са 38 цифара само првих седам су значајне. Имајући ово у виду довољно је да број PI дефинишемо у виду записа PI = 3,141592.

4.2. Логички подаци

Логички подаци се користе за представљање логичких израза који могу бити истинити (*true*) и неистинити (*false*). Природа ових података је целобројна па се вредност 0 тумачи као логичка неистина (*false*), а било која ненулта вредност као логичка истина (*true*). При дефинисању типа ових података у ARDUINO C програмима користи се назив *boolean*.

4.3. Подаци типа карактер

Подаци типа карактер (*char*) се користе за чување једног карактера у виду осмобитне бинарне комбинације. Појавом првих рачунара у оптицају је био мањи број карктера за чији опис су били довољни само седам бита. Имајући у виду стандард уведен у виду осмобитне речи (бајт) временом је овај скуп проширен са ограниченим бројем графичких симбола. Скуп карактера који су дефинисани користећи само осам бита познат је под називом ASCII (*American Standard Code for Information Interchange*). Подаци типа (*char*) припадају опсегу од -128 до +127. У случају када предзнак није битан треба користити податке типа (*unsigned char*) који припадају опсегу од 0 до 255. Очигледно, заузеће меоријског простора остаје исто (један бајт). Употреба ознаке *unsigned char* има смисла јер се тиме наглашава да су подаци позитивни цели бројеви чије вредности нису веће од 255.

4.4. Подаци типа бајт

Ови подаци заузимају исту величину меморијског простора као и подаци типа карактер (један бајт) с тим да не постоји бит за означавање предзнака вредности. Сходно томе, њихова примена је погоднија у односу на целобројне податке у ситуацији када они попримају вредности у интервалу од 0 до 255 јер заузимају мање меморијског простора (целобројни захтевају два бајта).

4.5. Подаци типа string

Овај тип података није ништа друго него секвенца ASCII карактера, повезаних у низ, који се третирају као јединствена целина, односно као један ентитет. При томе овај ентитет добија јединствено име. Другим речима, један или више података се појављују као уређени скуп са јединственим

именом. Подаци типа *string* могу да се дефинишу на неколико различитих начина. Уколико применимо опис у виду наредбе:

```
char myString[15];
```

подразумева се да је на овај начин додељен меморијски простор за смештање податка типа *string* са четрнаест карактера у себи. Приметимо да је ово за један мање у односу на број који је наведен у заградама. Разлог је веома једноставан. Програмски језик захтева да се на крају дода нул карактер '\0' како би компајлер могао да препозна крај низа. Због тога је димензија сваке *string* променљиве одређена бројем у заградама који је умањен за један. У конкретном случају, ово значи да је могуће користити *string* податак са највише 14 карактера. Програмска подрша је креирана тако да ће при дефинисању *string* података увек бити унет и нул карактер на крају. Следи неколико примера правилно дефинисаних *string* података:

```
char name[] = "Jana";
char name[5] = "Jana";
char name[100] = "Jana";
```

У првом примеру компајлер ће самостално открити колико је бајта потребно за смештај податка. С обзиром да име Jana садржи четири карактера, компајлер аутоматски додаје још један бајт за смештај нул карактера. У другом примеру је априорно дефинисана права величина меморијског простора па компајлер само уписује нул карактер на позицији последњег бајта. Наредба у последњем примеру омогућава да се унутар прва четири бита сместе карактери који одговарају имену „Jana“ додајући нул карактер на позицији петог бита како би се означио крај конкретног *string* податка. Имајући у виду бројну вредност унутар заграда у овом примеру јасно је да се име може проширити до 99 карактера.

Уочимо да се, при дефинисању, податак наводи између знакова навода.

Иницијализација *string* податка се може извршити, уколико је то из неког разлога погоднije, тако што се сваки карактер наводи посебно између апострофа и одваја запетом од претходног као у примерима, који следе:

```
char name[] = { 'J', 'a',  'n', 'a', '\0'};
char name[5] = { 'J', 'a',  'n', 'a', '\0'};
char name[] = { 'J', 'a',  'n', 'a'};
char name[5] = { 'J', 'a',  'n', 'a'};
```

Приметимо, као и раније, да је компајлер довољно „паметан“ да сам дода нул карактер тамо где је то потребно. У случају када се, при иницијализацији, сваки карактер наводи одвојено листа карактера започиње отвореном витичастом заградом ({) и завршава затвореном витичастом заградом (}).

4.6. Подаци типа String

Подаци типа *String* се разликују од података типа *string.* Они су произашли као надградња у односу на *string* податке. Отуда су названи истим именом уз примену великог слова на почетку. Међутим, за разлику од једноставних низова карактера они се могу третирати као објекти што нуди низ погодности при њиховој обради. Уколико претпоставимо постојање *string* променљиве под именом myData (низ карактера) и желимо да их преведемо у облик где ће сва мала слова бити замењена великим морали би да дефинишемо цео поступак као низ одговарајиућих наредби унутар конкретног програма. Међутим, уколико се дефинише да променљива myData има тип *String* тада овај посао постаје веома једноставан. Довољно је да само унесете наредбу:

```
myData = myData.toUpperCase();
```

и овај посао је, за вас као програмера, готов. Разлог је врло једноставан. С обзиром на природу овог типа података дефинисана је функцијска подршка у виду готовог потпрограма која овај посао чини једноставним. Ви једино треба само да дефинишете променљиву исписом наредбе:

```
String myData = String(100);
```

која резервише меморијски простор за смештање 99 карактера. Уочимо да је за употребу уграђених функција, у случају овог типа променљиве, неопходно навести тачку након имена променљиве (у овом случају она представља оператор), а након ње име конкретне функције (у овом случају toUpperCase). У случају да велика слова треба заменити малим треба поступити на истоветан начин само што је тада неопходно навести функцију toLowerCase:

```
myData = myData.toLowerCase();
```

Mada ARDUINO C није објектно оријентисани програмски језик постојање овог типа података понекад може бити веома корисно. Комплетан преглед расположивих функција дат је у табели 4-3 са описом на енглеском језику (ово је намерно учињено јер су имена функција у директној вези са овим описом).

Табела 4-3

Функција	**Намена**
String()	Define a String object.
charAt()	Access a character at a specified index.
compareTo()	Compare two Strings.
concat()	Append one String to another String.
endsWith()	Get the last character in the String.
equals()	Compare two Strings.
equalsIgnoreCase()	Compare two Strings, but ignore case differences.
getBytes()	Copies a String into a byte array.
indexOf()	Get the index of a specified character.
lastIndexOf()	Get the index of the last occurrence of a specified character.
length()	The number of characters in the String, excluding the null character.
replace()	Replace one given character with another given character.
setCharAt()	Change the character at a specific index.
startsWith()	Does one String begin with a specified sequence of characters?
substring()	Find a substring within a String.
toCharArray()	Change from String to character array.
toLowerCase()	Change all characters to lower case.
toUpperCase()	Change all characters to upper case.
trim()	Remove all whitespace characters from a String.

4.7. Void тип података

У литератури се void појављује у листи података међутим може се рећи да је ово пре службена реч која се користи при дефинисању функцијских потпрограма како би се нагласило, односно описало (дескриптор) да нема преноса променљивих из основног програма у конкретан потпрограм или из њега у основни програм. Пример употребе овог оператора налазимо у сваком ARDUINO програму (блокови наредби који започињу описом void setup() { } и void loop() { }).

4.8. Низови података (array)

Практично сви основни типови података могу се користити за формирање низова (array). Већ смо навели примере низова карактера. Примери који следе приказују поступак дефинисања низова у случају када њихови елементи нису ASCII карактери:

```
int myData[15];
long rfMeasurements[8];
float current[100];
```

Више детаља о примени низова биће изложено у поглављу о показивачима.

4.9. Дефинисање и декларисање промнљивих

Обе ове речи често се помињу у свету програмера као синоними. Међутим, дефинисање променљиве и њено декларисање имају заправо различито значење. Појаснимо разлику између ова два појма анализирајући шта се дешава у процесу компајлирања следеће наредбе:

```
int resADconv;
```

Наиласком на ову наредбу компајлер прво проверава да ли постоји нека грешка у погледу синтаксе. У случају такве грешке појавиће се поруке које описују о каквој грешци се ради. Ако грешке нема, проверава се да ли у листи имена, за конкретни програм, већ постоји променљива са именом resADconv. У случају да ово име није већ додељено некој променљивој наставља се са провером да ли је позната адреса меморијске локације на коју треба да буде смештен податак са именом resADconv . Ова адреса је названа lvalue (*location value*) и означава почетак меморијског простора у којем се смешта конкретни податак. У конкретном случају

ради се о целобројном податку који захтева простор величине два бајта. Обично се бајт са мањом тежинском вредношћу (првих осам бита читано с десна у лево) смешта на овој локацији, а бајт веће тежине на прву наредну меморијску локацију. Вредност која се смешта унутар конкретног меморијског простора назива се rvalue (*register value*).

Уколико за конкретан податак (resADconv), у листи симбола, постоји lvalue, односно постоји јасно дефинисана адреса меморијске локације на коју се смешта кажемо да је променљива дефинисана. Уколико адреса ове локације није позбната онда је променљива само декларисана.

4.10. Конверзија података из једног типа у други

Под конерзијом типа (*type cast*) подразумева се претварање података неког типа у еквивалентан податак другог типа. Унарни оператор за конверзију типа (*cast* оператор) чине симболи у облику отворене и затворене заобљене заграде. Између њих се наводи назив типа у који се податак претвара. На пример, уколико желимо да претворимо целобројни податак dataC у податак dataB типа byte потребно је написати наредбу

```
dataB = (byte) dataC;
```

Анализирајмо кратак програм у којем су иницијално дефинисане вредности четири различита типа података (int a, unsigned int b, float c, byte d):

```
// inicijalizacija tipa vrednosti podataka.
    int a = B1000001;
    unsigned int b = 0x41;
    float c = 65.;
    byte d = 0101;
    char dataC;
```

```
void setup() {
      // inicijalizacija serijskog prenosa za prikaz na monitoru
      Serial.begin(9600);
      // konverzija binarno definisanog podatka
      dataC = (char) a;
      Serial.println(dataC);
      delay(1000);                    // wait for a second
      // konverzija heksadecimalno definisanog podatka
      dataC = (char) b;
      Serial.println(dataC);
      delay(1000);                    // wait for a second
      // konverzija dekadsi definisanog podatka
      dataC = (char) c;
      Serial.println(dataC);
      delay(1000);                    // wait for a second
      // konverzija oktalno definisanog podatka
      dataC = (char) d;
      Serial.println(dataC);
      delay(1000);                    // wait for a second
}

// the loop function runs over and over again forever
void loop() {
}
```

Након извршења програма, сва четири податка су преведена у податак типа char. При томе се сваки пут на екрану монитора приказује новодобијена вредност.

4.11. Сажетак

Сваки податак има особине које одређују његову припадност неком од типова података. Тип податка одређује опсег вредности које може да узме (самим тим и заузеће меморије, односно број неопходних меморијских локација за његово смештање) као и операције које могу да се изводе над податком. Типови података представљају кључне речи у језику ARDUINO C. Појам кључна реч односи се на било коју реч која има специјално значење за C компајлер. Због тога се ове речи не могу користити за неке друге сврхе

Подаци могу да се представе помоћу вредности или помоћу идентификатора.

При именовању променљивих и функцијских потпрограма могуће је користити: сва слова енглеске абецеде, мала (a - z) и велика (A – Z), доњу црту (_) и цифре 0 до 9, али не као први знак у имену. Све остало није допуштено. У случају када се име састоји од више речи погодно је сваку реч започети великим словом.

Бројчани типови података се појављују као целобројни (*integer*) и реални (*real*) тип. Целобројни тип података омогућава појављивање бројева у облику позитивних или негативних целих вредности. Реални тип података подразумева могућност записа разломљених бројева у облику mEk, где се m назива мантиса, а k експонент. Велико слово E означава декадску бројну базу. Мантиса је децимални број који може да има предзнак, а целобројни и децимални део се обавезно раздвајају тачком. Експонент је цео број који може да садржи и предзнак.

Логички подаци се користе за представљање логичких израза који могу бити истинити (*true*) и неистинити (*false*).

Подаци типа карактер (*char*) се користе за чување једног карактера у виду осмобитне бинарне комбинације.

Подаци типа бајт заузимају исту величину меморијског простора као и подаци типа карактер (један бајт) с тим да не постоји бит који носи информацију о знаку.

Секвенца ASCII карактера, повезаних у низ, који се третирају као јединствена целина, представља *string*. При томе овај ентитет добија јединствено име. Другим речима, један или више података се појављују као уређени скуп са јединственим именом.

Подаци типа *String* се разликују од података типа *string.* Они су произашли као надградња у односу на *string* податке. Отуда су названи истим именом уз примену великог слова на почетку. Међутим, за разлику од једноставних низова карактера они се могу третирати као објекти.

Практично сви основни типови података могу се користити за формирање низова (array).

Службена реч void се користи при дефинисању функцијских потпрограма како би се нагласило, односно описало (дескриптор) да нема преноса променљивих из основног програма у конкретан подпрограм или из њега у основни програм.

Под конверзијом типа (*type cast*) подразумева се претварање података неког типа у еквивалентан податак другог типа. Унарни оператор за конверзију типа (*cast* оператор) чине симболи у облику отворене и затворене заобљене заграде. Између њих се наводи назив типа у који се податак претвара.

Питања

Како треба дефинисати име и шта је могуће користити при именовању променљивих?

1. Шта одређује тип податка?
2. Како се могу представити подаци?
3. Шта су структурирани подаци?
4. Шта представљају кључне речи у случају програмског језика?
5. Наведите и објасните типове бројчаних података.
6. Шта су логички, а шта подаци типа карактер?
7. У чему је разика између података типа *string* і *String*?
8. Шта представљају дефинисање и декларисање променљивих?

5

ОПЕРАТОРИ

Ово поглавље садржи опис оператора који се користе у развоју програмске подршке, њихови међусобни односи у погледу приоритета као и примери за илустрацију њиховог коришћења.

Имајући у виду да је у трећем поглављу већ било речи о аритметичким операторима и њиховим приоритетима јасно је да оператори представљају радње које се извршавају над операндима. Комбиновањем оператора и операнада настају изрази који, по потреби, могу бити записани унутар пара заобљених заграда. Њихова улога је да издвоје део сложеног израза као мању целину и тиме утичу на редослед извршења оператора. Већ је речено да оператори, у зависности од броја операнада на које се примењују, могу бити унарни, бинарни и тернарни.

У групи аритметичких оператора интересантно је приметити да се + и – појављују као унарни оператори када одређују предзнак неке вредности. С друге стране, када повезују два операнда у неком изразу они представљају бинарне операторе.

Унутар С језика уведени су специфични унарни оператори ++ и -- којим се омогућава промена вредности операнда за 1. При томе треба истаћи да се ови оператори могу појавити као префиксни, када се појављују испред операнда (++k, --k), или постфиксни када се појављују иза операнда (k++,k--).

У случају префиксне примене ових оператора у неком изразу, на пример (++k), током израчунавања се прво мења вредност операнда тако да у израчунавању израза учествује вредност операнда, (k = k+1). Аналогно правило важи у случају оператора -- само што тада у израчунавању израза учествује вредност операнда (k = k-1).

У случају примене постфиксног оператора унутар неког израза, на пример (k++), редослед израчунавања је промењен. Прво се рачуна вредност израза користећи непромењену вредност операнда k, а затим се мења вредност операнда (k = k+1). Аналоган процес дешава се и у случају оператора – па је тада коначна вредност операнда (k = k-1).

Наведимо пар једноставних примера:

```
++ k                    // k = k+1
k++                     // k = k+1
j= --k                  // k = k-1, j=k
j= k--                  // j=k,    k = k-1
```

Прва два израза показују да у случају самосталног израза префиксна и постфиксна варијанта оператора ++ имају исти ефекат. Претпоставимо да је почетна вредност k = 8, тада је нова вредност променљиве j у трећем изразу 7, а у четвртом 8. Нова вредност променљиве k, у оба случаја, је 7. Приметимо да су у последња два примера добијене две нове вредности помоћу једног израза.

5.1. Оператори за доделу вредности

Стандардан облик овог оператора (=) подразумева да се вредност израза на десној страни додељује, односно смешта у меморијски простор који је резервисан за чување вредности операнда на левој страни. Међутим ако погледамо табелу 5.1 уочићемо да се поред стандардног облика оператор (=) појављује и у комбинацији са другим операторима: +=, - =, *=, /=, %=, &=, ^=, |=, <<= и >>= који се уобичајено називају сложени оператори (*compound operators*). Улога појединих оператора биће описана касније.

Табела 5.1 садржи преглед оператора, у складу са њиховим приоритетом. При томе је највиши приоритет одређен бројем 1. Очигледно, оператори за доделу вредности имају најнижи приоритет. Наведимо неколико примера:

```
y = a * x + b           // y = ((a * x) + b)
d *= e + f              // d = (d * (e + f)
d = d * e + f           // d = ((d * e) + f)
a = b = c = d + 5       // a = (b = (c = (d + 5)))  →
                        // c = d + 5, b = c, a = b
a = b++ + 3*(c = d - 3) // a = (b++) + (3*(c = (d - 3)))  →
                        // c = (d – 3), u = b, b = b + 1, a = u + (3 * c)
```

Табела 5.1

Приоритет	Оператор
1	() [ ] → . (dot)
2	! ~ ++ -- +(unary) -(unary) *(indirection) &(address of) (cast) sizeof
3	* (multiplication) / %
4	+ (binary) - (binary)
5	<< >>
6	< <= > >=
7	== !=
8	& (bitwise AND)
9	^
10	|
11	&&
12	||
13	?:
14	= += - = *= /= %= &= ^= |= <<= >>=

Први израз представља уобичајени пример доделе вредности који добро познајемо из математике.

Други израз показује коришћење сложеног оператора за доделу вредности (*=). Због нижег приоритета овог оператора, операнд d, на левој страни, множи се резултатом целокупног израза на десној страни (e+f), без обзира што оператор сабирања има нижи приоритет од уобичајеног оператора множења (*). У случају када је потребно да се почетна вредност променљиве d помножи само са e и да се f дода на тај производ, израз треба да се пише на начин који је приказан у трећем примеру.

Четврти пример приказује могућност доделе исте вредности (d+5) већем броју променљивих помоћу једног

израза. С обзиром да се оператори за доделу вредности групишу с десна на лево, прво ће се вредност израза (d+5) доделити променљивој c, након тога променљивој b и коначно променљивој a. Истакнимо да је додела вредности већем броју променљивих у истом изразу могућа у случају језика C захваљујући томе што се додела вредности сматра операцијом, а не наредбом

5.2. Релацијски оператори

Операнди релацијских оператора су произвољног бројчаног типа (цели или реални), а резултат је логичког типа што значи да може имати вредности 1 (ако релација важи) или 0 (када релација не важи). Те вредности су формално boolean типа, али због своје целобројне природе могу да се користе и као операнди у аритметичким изразима.

Релацијски оператори (<) мање, (<=) мање или једнако, (>) веће, (>=) веће или једнако имају виши приоритет (6) у односу на операторе (==) једнако и (!=) различито чији је приоритет (7). Сви релацијски оператори се групишу с лева на десно. Анализирајмо неколико израза са релацијским операторима:

```
5>7              // 0

10<= 20          // 1

8 == 13>5        // 8 ==(13>5)          → 8 == 1          → 0

14 > 5 < 3       // (14 > 5) < 3        → 1 < 3           → 1

a < b < 5        // (a < b) < 5         → (0 или 1) < 5   → 1
```

Прва два израза су просте релације и не захтевају додатни коментар.

Следећа три оператора показују неке неуобичајене изразе са релацијским операторима. Приликом тумачења ових израза стално треба имати у виду приоритете оператора, целобројну природу логичких вредности као и чињеницу да релацијски оператори увек дају вредности 0 или 1.

5.3. Логички оператори

Овој групи припадају опертори логичко И (&&), логичко ИЛИ (||) и логичко НЕ (!). Ови оператори могу да се користе при дефинисању услова на основу којих се реализује процес одлучивања.

Претпоставимо да су дефинисана два оператора x и y. Резултат који настаје испитивањем услова

```
if (x == HIGH  && y == HIGH) { // Are both the TRUE?
  // ...
}
```

биће TRUE само ако су оба оператора TRUE.

У следећем примеру (логичко ИЛИ) резултат ће бити TRUE само уколико је бар једна од оператора, x или y већи од нула.

```
if (x > 0 || y > 0) {
  // ...
}
```

Коначно, у последењем примеру (логичко НЕ) резултат ће бити TRUE уколико је оператор x=0. У случају било које позитивне вредности резултат је FALSE.

```
if (!x) {
  // ...
}
```

5.4. Бит по бит (bitwice) оператори

При раду са битима могуће је користити операторе који као резултат генеришу вредности 1 или 0. Ове операције се могу обавити како над појединачним битима тако и на целој групи бита (на пример свих осам бита података типа char). Оператори ове групе су:

- & операција И са битима,
- | операција ИЛИ са битима ,
- ^ операција ексклузивно или са битима (XOR),
- ~ инверзија (комплементирање) бита,
- << померање бита за једно место у лево,
- >> померање бита за једно место у десно.

5.4.1. Примери операција са битима

За илустрацију операција са битима претпоставимо да су дефинисана два операнда са бинарним вредностима B1000001 и B1000000. Ово су заправо бинарни еквиваленти вредности којима се у складу са ASCI стандардом представљају велика слова латинице A и B. Реализацијом прграмске секвенце:

```
byte op1 = B1000001;
byte op2 = B1000010;
byte rzlt;
void setup() { // put your setup code here, to run once:
        Serial.begin(9600);
        rzlt = op1 & op2;
        Serial.print("op1 & op2 = "); Serial.println(rzlt);
}
```

добија се резултат у виду исписа броја 64. Ово је заиста декадски еквивалент бинарног броја B100 0000 који означава симбол @. Шта се заправо догодило? Испишимо вредности операнада као и резултат на следећи начин:

карактер	бинарно	декадски
A	100 0001	65
B	100 0010	66
@	100 0000	64

Очигледно, извршена је И операција између одговарајућих бита оба операнда (оба одговарајућа бита морају бити 1 да би резултат био 1).

Уколико се, у приказаној програмској секвенци, И оператор (&) замени оператором ИЛИ (|) добиће се следећи резултат:

карактер	бинарно	декадски
A	100 0001	65
B	100 0010	66
C	100 0011	67

Очигледно ИЛИ операција над битима карактера чије вредности представљају велика слова A и B даје, као резултат, карактер C.

Уколико се, у приказаној програмској секвенци, И оператор (&) замени оператором ЕКСКЛУЗИВНО ИЛИ (^) добиће се следећи резултат:

карактер	бинарно	декадски
A	100 0001	65
C	100 0010	66
	000 0011	3

Очигледно ЕКСКЛУЗИВНО ИЛИ операција обезбеђује да се у резултату појави јединица само ако је један од упоређених битова 1, а други 0. У случају да оба бита имају вредности 1 резултат ће бити 0.

Примена оператора за комплементирање (~) над неким операндоим има за последицу промену вредности сваког појединог бита (комплементирање бита).

Операције за померање бита << (у лево) и >> (у десно) изводе се тако што се садржај операнда помера у лево или десно (зависно од примењеног оператора) за број места који је одређен вредношћу броја наведеног иза оператора. Тако у случају наредби:

```
byte a=5;          // a ima binarnu vrednost 00000101
byte b=a<<3;       // b ima binarnu vrednost 00101000
```

садржај операнда b чини вредност која је добијена померањем садржаја операнда a за три места у лево. При томе су упражњена места попуњена са цифрама 0. Очигледно померањем за три места у лево добија се вредност (b=40) која је за 2^3 пута већа у односу на претходну вредност (a=5). Операција померања бита се управо често користи за једноставна целобројна множења јер су много бржа у односу на операцију множења целих бројева. Поред тога, у комбинацији са И/ИЛИ операторима применом померања бита могуће је реализовати селективно очитавање појединих бита као на пример у случају серијског преноса података.

5.5. Оператори за рад са показивачима

Оператори који омогућавају примену показивача су * и &. Уочимо да су ови симболи већ коришћени за дефинисање других оператора. На први поглед делује нелогично да се могу користити за ознаку неке друге операције. Као што смо већ видели && и & означавају два различита оператора. Тако и

симболи * и &, у зависности од тога како су позиционирани у односу на суседне симболе, имају другачије значење, односно могу представљати другачије операторе. Њихово дефинисање и правилна употреба биће детаљно описани у поглављу о показивачима.

5.6. Сажетак

Специфичност ARDUINO C програмског језика је да нема сва својства стандардног C језика. Постојање префиксних и постфиксних оператора (++ или --) као и сложених оператора (+=, - =, *=, /=, %=, &=, ^=, |=, <<= и >>=) у значајној мери може да поједностави писање програма, при чему треба добро водити рачуна о споредним ефектима које примена ових оператора производи. Поред тога посебно треба водити рачуна о приоритету оператора како се не би добили резултати мимо онога што се заправо жели.

Питања

1. Наведите префиксне и постфиксне операторе и објасните ефекте који се појављују при њиховој примени.
2. Наведите сложене операторе и објасните ефекте који се појављују при њиховој примени.
3. Који опеартор има највиши а који најнижи приоритет?
4. Објасните разлику између логичких и релацијских оператора.
5. Која вредност се добија након извршења наредби

```
int a= 2;  int b=a<<8; ?
```

6

УПРАВЉАЧКЕ СТРУКТУРЕ

Поглавље започиње навођењем основних управљачких структура након чега следи детаљан опис сваке од њих. Поред тога дати су и примери којим се илуструје начин коришћења као и ефекти примене одговарајуће структуре на ток програма.

Највећи део неког програма реализује се узастопним извршењем једноставних наредби (секвенца). Промена тока програма се остварује извршењем неке од управљачких наредби које у складу са вредностима појединих променљивих могу да доведу до тога да се делови програма прескоче (селекција) или да се њихово извршење понови више пута (циклус или петља).

Управљачке наредбе, очигледно, не врше никакву обраду већ само преносе ток управљања на неки део програма. Поред селекција и циклуса у ову групу наредби спадају и наредбе за реализацију скокова.

6.1. Селекције

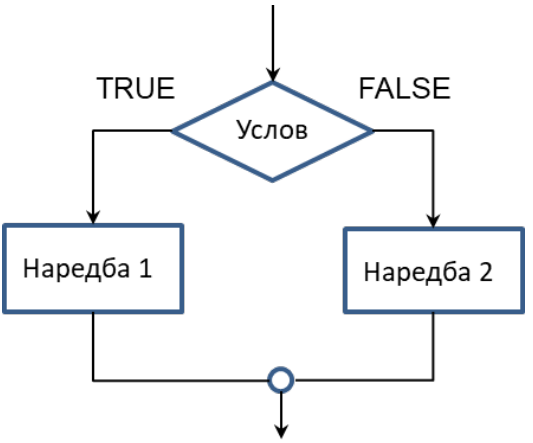
Селекције су управљачке структуре које омогућавају условно извршавање наредби. Оне, за извршавање, изаберу једну од наредби из скупа од једне, две или више наредби. Зависно од услова бирања, који је део селекције, може да се деси да ни једна наредба не буде извршена. Наредбе између којих се врши избор могу да буду како просте тако и сложене, што значи да избор може да буде и између неколико сложених обрада.

Основни облик if-else селекције формира се у виду записа:

```
if (uslov)
        naredba 1;      // ili blok naredbi
else
        naredba 2;      // kraj if-else selekcije
```

Уколико је испуњен наведени услов (одговара стању TRUE) извршиће се наредба 1. У супротном случају, извшава се наредба 2. У оба случаја извршава се само једна наредба па није неопходно да се оне смештају између витичастих

заграда. Дијаграм тока који одговара овом запису приказан је на слици 6.1.



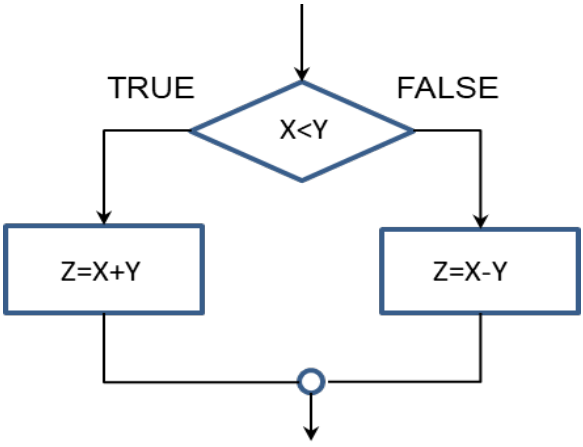
Слика 6.1 if-else структура

Пример који следи илуструје основни облик if-else селекције:

```
void setup() { // put your setup code here, to run once
        Serial.begin(9600);
        unsigned char x, y, z;
        char simbol;
        x= 13;
        y= 52;
        if (x < y)
                z = x + y;
        else
                z = x - y;
        simbol = z;
        Serial.println(z);
        Serial.println(simbol);
}
void loop() {
        // put your main code here, to run repeatedly:
}
```

Уочимо да су најпре декларисане три променљиве (x, y, z) типа unsigned char и једна променљива (simbol) типа (char). Затим су додељене конкретне вредности (x= 13; y= 52;). Зависно од тога да ли је испуњен услов (x < y), вредност променљиве z биће израчуната као сума (x + y) или разлика (x - y).

У конкретном случају услов је испуњен па се вредност променљиве z израчунава као сума (x + y). У оба случаја, вредност променљиве z биће додељена, извршењем прве следеће наредбе иза if-else структуре, променљивој simbol. Након тога, на екрану монитора ће се појавити број 65, а затим испод њега велико слово A (уколико нисте сигурни зашто се ово дешава погледајте поново поглавље о типовима података). Слика која следи представља дијаграм тока који одговара управо описаном примеру.

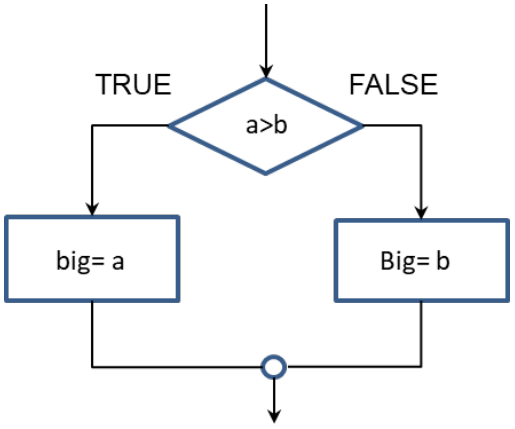


Слика 6.2 Пример if-else структуре

Следећи једноставан пример омогућава да се одреди који је од два осмобитна неозначена броја, a и b, већи, а затим смести у меморијски простор који одговара променљивој са именом big.

```
void setup() {
        // put your setup code here, to run once
        Serial.begin(9600);
        unsigned char a, b, big;
        a= 67;
        b= 66;
        if (a>b)
                big = a;
        else
                big = b;
        Serial.print(big);
}
void loop() {
        // put your main code here, to run repeatedly:
}
```

Слика 6.3 приказује дијаграм тока који одговара управо описаном примеру.



Слика 6.3 Пример if-else структуре за одређивање већег од два неозначена броја

Ово је право место да појаснимо како се користи тернарни оператор (?:). Управо у случају када треба да се изврши једна од две једноставне наредбе, уместо if-else структуре, може да се употреби тернарни оператор. Погледајмо следећи програм:

```
int a=3, b=5;
int x;
void setup() {
        // put your setup code here, to run once:
        Serial.begin(57600);
        (a>b) ? (x=a) : (x=b);
        Serial.print("Maksimalna vrednost je x = ");
        Serial.print(x);
}
void loop() {
        // put your main code here, to run repeatedly:
}
```

Очигледно, задатак програма је да одреди који од два понуђена броја је већи (a или b) и да испише одоварајућу поруку на екрану. У случају када је наведени услов испуњен извршава се наредба која следи одмах након знака питања и након тога програм прелази на извршење наредбе у следећем реду. Међутим, уколико наведени услов није испуњен извршава се наредба наведена иза две тачке и наставља извршење наредбе у следећем реду.

У случају када је, у зависности од испуњености услова (true or false), потребно да се изврши више наредби неопходно је да се формира одговарајући блок наредби тако што ће оне бити написане између витичастих заграда. Илуструјмо овај поступак једноставним примером. Нека су x и y два неозначена броја. Претпоставимо да је формулисан захтев у следећем облику:

1) Уколико је x>y, тада треба одредити вредности x+y и x/y и сместити их у меморијски простор као променљиве u и v

2) У супротном случају, потребно је израчунати x-y, а затим увећати променљиву w за 4 и сместити новодобијене вредности у меморијски простор као променљиве z и w.

3) Коначан корак, значи без обзира на то да ли је услов испуњен или не, представља израчунавање вредности x*y, и њено меморисање у виду променљиве t.

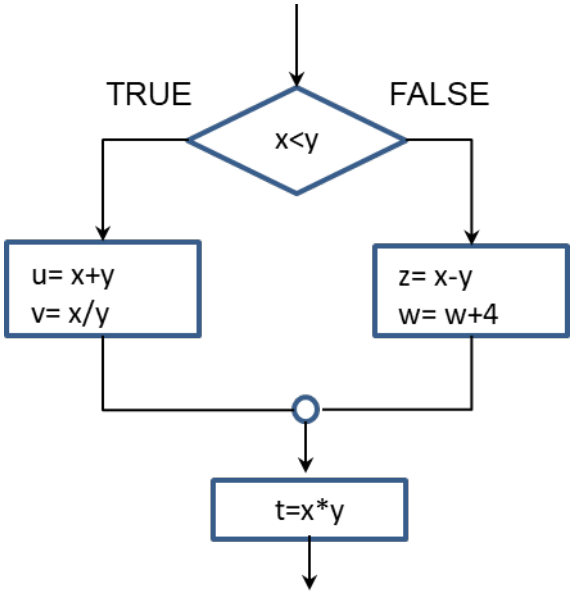
Следеће програмске линије илуструју овај поступак:

```
unsigned char t, u, v, w, x, y, z;
if (x < y) {
        u = x + y;
        v = x / y;
}
else {
        z = x-y;
        w+ = 4; // vrednost w se uvecava za 4
}
t = x*y
```

Видимо да су најпре све променљиве декларисане као подаци типа unsigned char. Уколико је испуњен услов (x < y) извршиће се наредбе којима се додељују вредности променљивама u и v, а затим ће се прећи на извршење наредбе t = x*y којом се додељује вредност променљиви t .

Међутим, уколико услов није испуњен извршиће се наредбе унутар блока који следи иза речи else којима се додељују вредности променљивама z и w, а затим као и у претходном случају прећи ће се на извршење наредбе t = x*y.

С обзиром да се наредба t= x*y извршава без обзира на испуњеност услова она се налази изван наведених блокова. Слика 6.4 приказује дијаграм тока који одговара овом примеру.



Слика 6.4 Пример if-else структуре са блоковима наредби

У пракси се појављују ситуације када је потребно променити ток програма само ако је постављени услов испуњен. У супротном случају програм наставља свој стандардни ток, односно нема потребе да се изврше неке додатне акције. На пример, ако је притиснут тастер треба да се промени статус одговарајуће светлеће диоде (ако је светлела треба је искључити и обрнуто). Тада нема потребе да се у структури која одговара селекцији типа if-else појављује реч else. Другим речима, постоји само један блок наредби (једна наредба уколико је она довољна за реализацију активности када је услов испуњен) након чега следи низ наредби чије извршење не зависи од наведеног услова.

Следе примери програма са селекцијом типа if-else. Извршењем сваког од ових програма обезбеђује се наизменично укључивање и искључивање две светлеће диоде у трајању од 1000ms.

Основна верзија програма:

```
/* Alternate Blink - Turns on one LED on for 1 second while the other is off, then reverses the LEDs for 1 second,repeatedly.*/

// Pin 13 has an LED connected on most Arduino boards. give it a name:
int led1 = 13;
int led2 = 12;                    // This is for the second LED

// the setup routine runs once when you press reset:
void setup() {
        // initialize the digital pin as an output.
        pinMode(led1, OUTPUT);
        pinMode(led2, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
        digitalWrite(led1, HIGH);// turn the LED on (HIGH is the voltage level)
        digitalWrite(led2, LOW); // turn the LED off by making the voltage LOW
        delay(1000);             // wait for a second
        digitalWrite(led1, LOW); // turn the LED off by making the voltage LOW
        digitalWrite(led2, HIGH);// turn the LED on (HIGH is the voltage level)
        delay(1000);             // wait for a second
}
```

Видимо да се, у овој верзији, циклично понављају процеси укључења и искључења светлећих диода унутар void loop блока програма.

У наредној верзији програм је модификован увођењем if-else структура и бројача counter. Током сваког проласка кроз

void loop блок програма садржај бројача counter се увећава за један. Унутар прве if-else структуре испитује се да ли је децимални остатак након дељења садржаја бројача са 2 један. Уколико је овај услов испуњен извршавају се наредбе унутар блока наредби које су смештене непосредно иза овог услова. Даљим извршењем програма прескаче се блок смештен унутар наредне if-else структуре и увећава садржај бројача counter за један.

Модификована верзија програма:

```
// the loop routine runs over and over again forever:
void loop() {
    if (counter % 2 == 1) {
        digitalWrite(led1, LOW); // turn the LED off by making the voltage LOW
        digitalWrite(led2, HIGH);// turn the LED on (HIGH is the voltage level)
        delay(1000);                    // wait for a second
    }
    if (counter % 2 == 0) {
        digitalWrite(led1, HIGH);// turn the LED on (HIGH is the voltage level)
        digitalWrite(led2, LOW); // turn the LED off by making the voltage LOW
        delay(1000);                    // wait for a second
    }
    counter = counter + 1;
}
```

Међутим, уколико је садржај бројача дељив са два, односно децимални остатак има вредност нула наведени услов (counter % 2 == 1) није испуњен па ће се прећи на испитивање наредног услова (counter % 2 == 0). Како је овај услов тада испуњен извршиће се наредбе унутар блока који припада овој if-else структури. Након тога се садржај бројача увећава за један.

Очигледно наведени услови испитују парност садржаја бројача counter. У зависности од тога да ли је паран или

непаран извршће се један или други блок наредби у сваком проласку кроз void loop() структуру.

Уочимо да је непотребно испитивати оба услова при сваком проласку кроз void loop() структуру. Уколико је испуњен један од наведених услова онда други сигурно није (ако је садржај бројача паран нема смисла тестирати да ли је непаран). Услед тога је могуће поједноставити програм, наводећи само један услов:

```
void loop() {
    if (counter % 2 == 1) {
        digitalWrite(led1, LOW); // turn the LED off by making the voltage LOW
        digitalWrite(led2, HIGH);// turn the LED on (HIGH is the voltage level)
     } else {
        digitalWrite(led1, HIGH);// turn the LED on (HIGH is the voltage level)
        digitalWrite(led2, LOW); // turn the LED off by making the voltage LOW
    }
    delay(1000);                    // wait for a second
    counter = counter + 1;
}
```

У наредној верзији програма проблем наизменичног укључења и искључења светлећих диода решава се тако што, у зависности од испуњености наведеног услова, променљиве led1, led2 добијају нове вреднсти, односно замењују им се додељене вредности.

```
void loop() {
    if (counter % 2 == 1) {
        led1 = 13;
        led2 = 12;
    } else {
        led1 = 12;
        led2 = 13;
    }
    digitalWrite(led1, HIGH);     // turn the LED on (HIGH is the voltage level)
```

```
    digitalWrite(led2, LOW);    // turn the LED on (HIGH is the voltage level)
    delay(1000);                // wait for a second
    counter = counter + 1;
}
```

Нагласимо да је примена if-else селекције могућа и у случају постојања више услова, односно у ситуацијама када током одлучивања могу настати више од два исхода. Тада структура if-else селекције има облик:

```
if (uslov1)
        naredba 1;
else if (uslov2)
        naredba 2;
else
        naredba 3;
```

Испитивање услова се врши редом, одозго на доле. Чим се наиђе на услов који је испуњен следи извршење наредбе која му припада. Након тога се излази из ове управљачке структуре и извршава прва следећа наредба .

У једноставном примеру који следи видимо да се вредност константе K мења у зависности од вредности променљиве x.

```
if (x >= 30)
        K = 1/2;
else if (x >= 10)
        K = 2;
else
        K = 3;
y = K*x;
```

У овом једноставном примеру видимо да се коефицијент пропорционалности мења у зависности од

величине променљиве x. Тако ће у случају да је *x* = 20 вредност *y* бити израчуната као *y*= 2*20.

6.2. Циклуси

Циклуси су управљачке структуре које омогућавају понављање наредбе или блока наредби. ARDUINO C подржава три типа циклуса: WHILE, FOR и DO. Добро функционисање било којег од наведених типова циклуса заснива се на испуњењу три услова.

Први услов одсликава захтев да се пре почетка циклуса дефинише вредност променљиве која се користи за контролу броја пролаза кроз циклус. Поред ове променљиве, у овом тренутку, по потреби се могу дефинисати и додатне променљиве. Очигледно, ова активност се дешава само једном, при уласку у циклус.

Други услов проистиче из потребе да се током сваког проласка проверава да ли у наредном кораку треба поновити циклус. Ова активност се реализује користећи релациони оператор у присуству контролне променљиве.

Последњи услов проистиче из потребе да се контролна променљива мења након сваког проласка како не би дошло до бесконачног понављања циклуса.

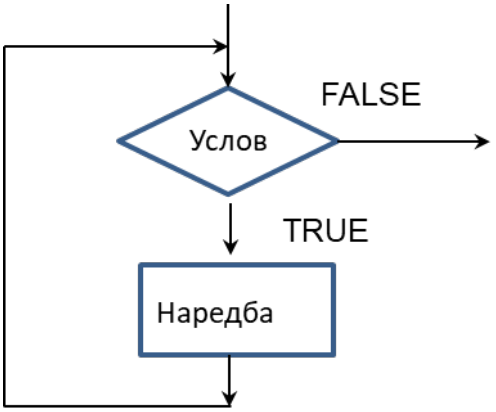
6.2.1. Циклус типа WHILE

Циклус типа WHILE представља управљачку структуру са излазом на врху и дефинише се користећи следећи опис:

```
while (uslov) {
naredbe
}
sledeća naredba;
```

Уколико се као резултат испитивања услова добија вредност која одговара логичком TRUE, извршиће се наредбе

које чине тело циклуса (наредбе између витичастих заграда), а након тога се поново прелази на проверу испуњености услова. Циклус се понавља све док је услов испуњен. Тада програм наставља извршење почевши од прве наредбе која следи иза тела циклуса. На слици 6.5 се види дијаграм тока који одговара циклусу типа WHILE.



Слика 6.5. Дијаграм тока који одговара циклусу типа WHILE

У примеру који следи тело циклуса се састоји од само једне наредбе па су витичасте заграде изостављене.

```
int n = 4;
while (n <= 16)
n + = 4;
z = n;
```

Уочимо да је најпре дефинисана променљива која се користи унутар циклуса. Њена почетна вредност је 4. Уласком у циклус проверава се да ли је вредност променљиве мања од 16. С обзиром да је услов испуњен прелази се на извршење наредбе, унутар циклуса, којом се постојећа вредност променљиве увећава за 4. Очигледно, циклус ће се понављати тако што променљива n поприма вредности 8, 12, 16, 20. У моменту када променљива n достигне вреност већу од 16 прећи ће се на извршење прве следеће наредбе којом

се променљивој z додељује достигнута вредност променљиве n (у овом случају 20).

Следећи пример приказује циклус којим се постиже кашњење у извршењу програма:

```
unsigned int k = 1000; // initialize k to 1000
while (k>10)
k --;
```

Тело цилуса чини само наредба којом се сваки пут вредност роменљиве k умањује за 1.

У следећем примеру приказан је угнежђени цилкус. Овом организацијом повећава се кашњење у извршењу програма. Главни циклус се извршава 1000 пута. Међутим, циклус који се налази унутар њега извршава се, у свакој итерацији, 99 пута. Тако је укупно кашњење вишеструко увећано. У конкретној ситуацији унутрашњи циклус се понавља 1000×99 пута пре него што се пређе на извршење прве следеће наредбе иза главног циклуса.

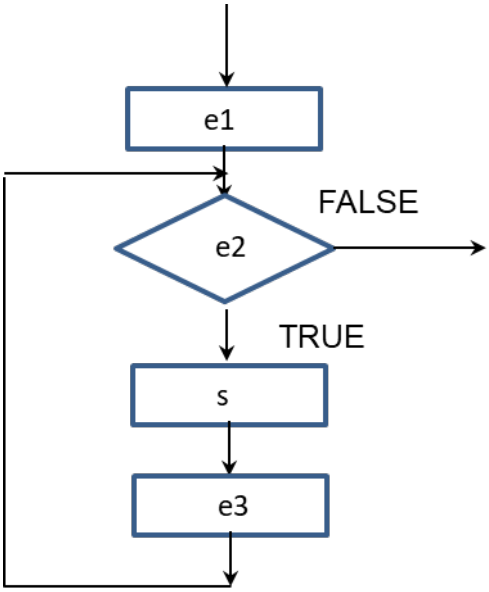
```
unsigned int i, j, k ;
i = 0; k = 1000;
while (i < k) {
j = 1;
while (j < 100)
j++;
i++;
}
```

6.2.2. Циклус типа FOR

Овај тип циклуса се чешће користи у односу на већ описани циклус типа WHILE. Његова синтакса има облик:

```
for (e1; e2; e3)
naredba;          // ili blok naredbi
```

При томе су e1, e2, e3 изрази. Почетна вредност одговарајуће променљиве одређује се на основу израза e1. Израз e2 представља услов који се проверава у циљу доношења одлуке о напуштању циклуса. Израз e3 обезбеђује ажурирање, односно промену контролне променљиве након извршења наредби унутар тела циклуса. Одговарајући дијаграм тока приказан је на слици која следи:



Слика 6.6. Дијаграм тока који одговара циклусу типа FOR.

Уколико је испуњен услов дефинисан изразом e2 извршиће се наредбе садржане у телу циклуса након чега се процес извршења програма усмерава на обраду израза e3 и поновну проверу услова дефинисаног изразом e2. Уколико се констатује да услов није испуњен циклус се не понавља већ се извршава прва наредба иза њега. Значи излаз из циклуса се остварује на његовом почетку, односно на врху.

Претходно приказани пример WHILE циклуса

```
int n = 4;
while (n <= 16)
```

```
n = n + 4;
z = n;
```

може да се опише користећи циклус типа FOR:

```
int n;
for (n = 4; n<= 16; n+ = 4)
z = n;
```

Поступајући на исти начин, већ приказани пример угнежђеног циклуса типа WHILE може да се опише користећи циклус типа FOR:

```
unsigned int i, j, k = 1000;
for (i = 0 ; i < k ; i++)
for (j = 0 ; j<100 ; j++)
```

Намеће се питање:„Зашто онда користити циклус типа WHILE?”. Замислимо ситуацију да у огромном броју података треба да проверимо да ли постоји неки конкретан податак. Уколико применимо циклус типа WHILE претраживање се прекида истог тренутка када се пронађе конкретан податак. У случају циклуса типа FOR претраживање се наставља све док се не достигне гранична вреднст дефинисана унутар услова за окончање циклуса. Очигледно, у зависности од проблема који се решава коритиће се један или други тип цилкуса.

Искористимо прилику да поново демонстрирамо могућност коришћења тернарног оператора (?:). Његовом применом могуће је дефинисати универзални, односно генерализовани циклус са изласком на врху. Ово је врло погодно у случају бројачких циклуса. Типичан пример овог циклуса је

```
for (i=pocetak; i<kraj; i+=korak) {...}
```

где pocetak и kraj представљају почетну и крајњу вредност бројача i a korak је износ промене бројача на крају сваког пролаза кроз циклус. У конкретном случају претпоставља се

да је korak већи од нуле. У случају негативног корака треба да се промени релацијски оператор < у >. Применом тернарног оператора може се дефинисати унверзални циклус који исправно функционише у оба случаја, односно како за негативну тако и за позитивну вредност корака. Анализирајмо следећи пример:

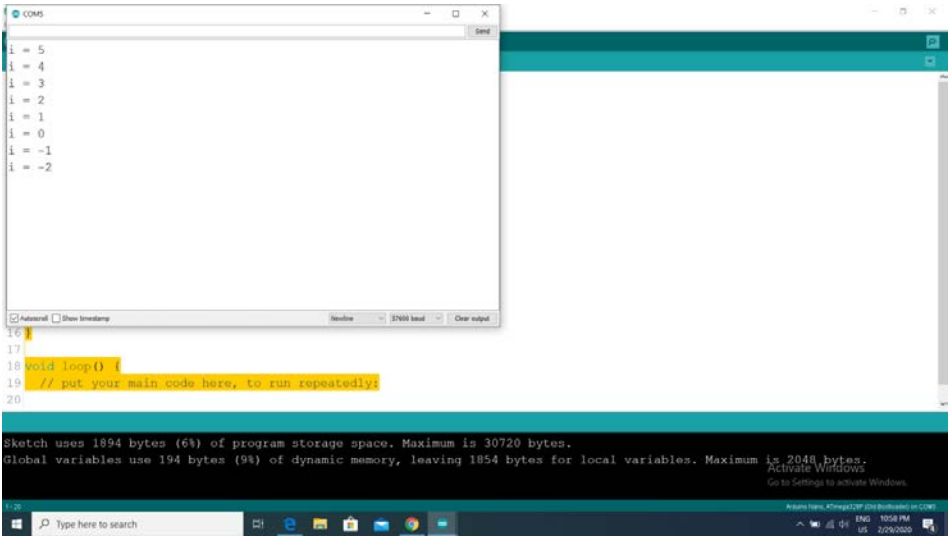
```
int pocetak=5, kraj=-2;
int korak= -1;
int i;

void setup() {
        // put your setup code here, to run once:
        Serial.begin(57600);

        // for (initialization; condition; increment)
        for(i = pocetak; (korak>0) ? (i<kraj):(i>kraj); i+=korak){
                Serial.print("i = ");
                Serial.println(i);
        }
        Serial.print("i = ");
        Serial.println(i);
}

void loop() {
        // put your main code here, to run repeatedly:
}
```

уочимо да су почетна и крајња вредност 5 и -2, а корак је негативан -1, односно има вреност -1. Извршењем овог програма на екрану серијског монитора добија се резултат који је приказан на слици 6.7.

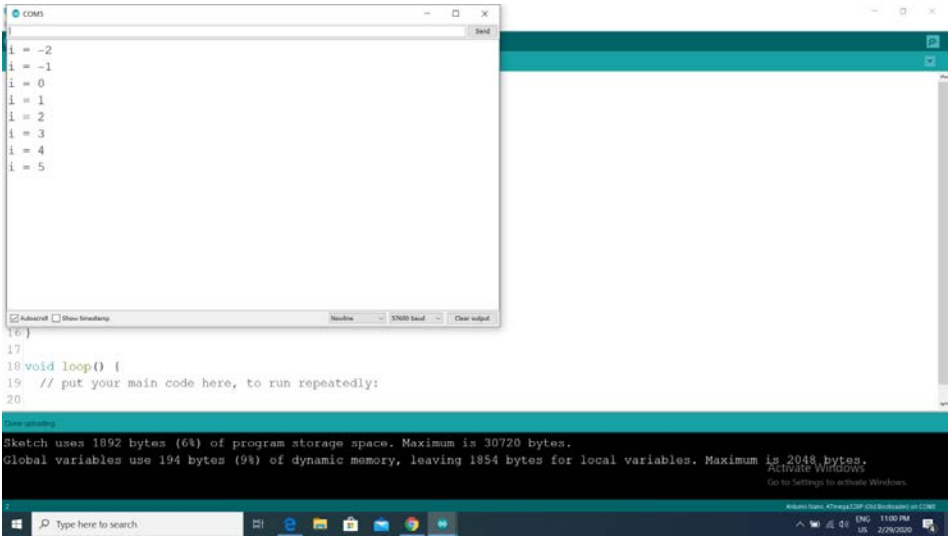


Слика 6.7. Резултат извршења програма у којем је FOR циклус дефинисан применом тернарног оператора (корак је негативан број).

У случају када се вредност контролне промељиве увећава, односно корак има вредност већу од нуле, на пример 1 мењају се само почетна и крајња вредност (почетна треба да је мања од крајње):

```
int pocetak= -2, kraj= 5;
int korak= 1;
int i;
```

Резултат извршења овог програма приказан је на слици 6.8.



Слика 6.8. Резултат извршења програма у којем је for циклус дефинисан применом тернарног оператора (корак је позитиван број).

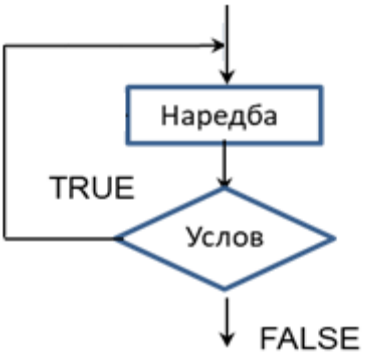
6.2.3. *Циклус типа DO-WHILE*

Полазећи од синтаксе овог цилкуса

```
do {
        naredbe
} while (uslov);
```

одмах можемо уочити да се наредбе унутар тела циклуса извршавају бар једном јер се услов за напуштање циклуса, за разлику од претходно описаних типова, проверава тек на крају. Дијаграм тока који одговара овом типу циклуса приказан је на слици 6.9.

Полазећи од чињенице да се било шта што је дефинисано овим циклусом може успешно описати применом циклуса типа FOR разумљиво је да се овај тип циклуса ретко користи у пракси.



Слика 6.9. Дијаграм тока који одговара циклусу типа DO – WHILE.

6.3. Селекција типа SWITCH

Селекција типа switch омогућава директно извршење наредбе или блока наредби у складу са вредношћу резултата који се добија као резултат израчунавања израза наведеног између заграда непосредно иза речи switch. Ова селекција се дефинише на следећи начин:

```
switch (izraz) { // otvorena zagrada za početak bloka switch naredbi
case 1:
	// naredbe koje se izvršavaju ako je vrednost izraza 1
break;
case 2:
	// naredbe koje se izvršavaju ako je vrednost izraza 2
break;
case 3:
	// naredbe koje se izvršavaju ako je vrednost izraza 3
break;
	// ukopliko je potrebno definišu se blokovi naredbi za ostale vrednosti
default:
	// naredbe koje se izvršavaju za slučaj vrednosti koje nisu naveden iza reči "case"
} // zatvorena zagrada za kraj bloka switch naredbi
// Ovo je prva naredba nakon bloka switch
```

У случају када се као резултат рачунања израза добије вредност која не постоји иза речи case тада се извршава блок наредби иза речи default. Резултат израза треба да буде податак типа byte, char, int или long (укључујући и unsigned варијанте).

Уочимо да блокови наредби који следе иза речи case нису коришћене витичасте заграде. Почетак блока наредби, у овом случају, компајлер препознаје наиласком на две тачке иза речи case, а крај наиласком на наредбу break. Извршењем наредбе break ток програма се преусмерава на прву наредбу иза витичасте заграде која означава крај switch блока.

Уколико на крају блока наредби који следи иза неке од case опција не постоји break наредба програм наставља да извршава наредбе у оквиру следеће case опције. Наравно, изостанак break наредбе, на крају одговарајућег блока, изазваће неочекивани ток програма само уколико смо заборавили да је напишемо. Међутим, може се десити да постоје наредбе које треба извршити у случају неколико различитих вредности. Одговарајући блокови наредби тада се намерно не завршавају са наредбом break. Да би ово разумели наведимо један једноставан пример . Претпоставимо да микроконтролер треба да преведе поруку која је дефинисана као податак типа string у сигнал којег чине морзеови знаци. Нека је то реч „BISER”. У таблици морзеових сигнала свако од ових слова је дефинисано као одговарајућа комбинација тачки (.) и повлака (-):

B (- . . .),
I (. .),
S (. . .),
E (.),
R (. - .).

Већ на први поглед уочавамо да су градивни елементи који чине слова S, I и E садржани у слову B. Уколико се порука прати преко светлеће диоде која је на ARDUINO платформама већ уграђена и повезана са пином 13 онда ће, у духу већ виђених примера, блокови наредби за формирање тачки и повлака имати следећи опис:

```
digitalWrite(LED_BUILTIN, HIGH);
delay(tSimbol);
digitalWrite(LED_BUILTIN, LOW);
delay(tPauza);
```

При томе су tSimbol трајање симбола који одговара тачки, односно повлаци, а tPauza пауза између емитовања два узастопна симбола (тачка или повлака). Нагласимо да важе следећи односи:

- трајање повлаке је три пута дуже од тачке,
- пауза између узастопних симбола траје исто колико и тачка,
- пауза између слова унутар исте речи траје као две тачке и
- коначно пауза између речи одговара трајању седам тачки.

Опис сваког слова би захтевао вишеструко понављање блока ових наредби уз дефинисање одговарајућих времена. Једноставније је, уколико ово учинимо на почетку програма, дефинишући одговарајуће макрое.

Шта је заправо макро? Ништа друго него име које се додељује групи наредби које се често понављају у писању прогама. Програмски језик C подразумева да се при дефинисању макроа користи синтакса:

```
#define imeMakroa {blok naredbi ispisanih u istom redu}.
```

У конкретном случају, генерисање тачке биће омогућено применом следећег макроа:

```
#define dot {digitalWrite(LED_BUILTIN, HIGH); delay(td);
        digitalWrite(LED_BUILTIN, LOW); delay(td);}
```

Уочимо да је име овог макроа dot (тачка). При томе td означава њено трајање. На овај начин писање програма се поједностављује јер сваки пут кад треба обезбедити генерисање симбола који одговара тачки само треба навести назив макроа (у овом случају dot). При компајлирању ће свака линија програма на којој се појави име овог макроа аутоматски бити замењена кодом који одговара наредбама у његовом саставу. Очигледно, примена макроа омогућава лакше писање и прегледнију форму изворног програма.

Аналогним поступком дефинише се макро са именом dash који треба да обезбеди генерисање повлаке:

```
#define dash {digitalWrite(LED_BUILTIN, HIGH); delay(tds);
        digitalWrite(LED_BUILTIN, LOW); delay(td);}
```

При томе tds означава трајање повлаке. С обзиром да након задњег симбола унутар кода за неко слово следи два пута дужа пауза (у програму је означена именом trs) дефинисана су још два аналогна макроа dotEnd (за случај када је тачка последња у низу симбола који дефинишу код за конкретно слово) и dashEnd (за случај када је повлака последња у низу симбола који дефинишу код за конкретно слово).

Напоменимо да словима B, I, S, E и R, у складу са стандардом ASCII, одговарају декадски бројеви 66, 73, 83, 79 и 82. Ови бројеви морају да буду написани након речи case на местима где започиње опис поступка за генерисање морзеовог кода који одговара конкретном слову. Конкретан програм чине следеће линије изворног кода:

```
/* Namena: Automatsko generisanje Morzeovog koda za poruku koju sadrži promenljiva tipa string */
#define dot {digitalWrite(LED_BUILTIN, HIGH); delay(td);
                digitalWrite(LED_BUILTIN, LOW); delay(td);}
#define dotEnd {digitalWrite(LED_BUILTIN, HIGH); delay(td);
                  digitalWrite(LED_BUILTIN, LOW); delay(trs);}
#define dash {digitalWrite(LED_BUILTIN, HIGH); delay(tds);
                digitalWrite(LED_BUILTIN, LOW); delay(td);}
#define dashEnd {digitalWrite(LED_BUILTIN, HIGH); delay(tds);
                    digitalWrite(LED_BUILTIN, LOW); delay(trs);}
int td = 200, tds = 3 * td, trs = 2 * td;
int i = 0, j, k;
char msg[] = "BISER-"; // Sadrzaj PORUKE
void setup() {
    pinMode(LED_BUILTIN, OUTPUT); // init. pin LED_BUILTIN as an output.
    k = sizeof(msg); // određuje se veličina poruke, odnosno broj slova
}
void loop() {
    j = i % k;
    switch (msg[j]) {
        case 82:                    // R
                dot
                dash
                dotEnd
        break;
        case 66: dash               // B
        case 83: dot                // S
        case 73: dot                // I
        case 69: dotEnd             // E
            break;
        default: delay(5 * trs);
    }
    i++;
}
```

Одмах, након уводног коментара следе дефиниције већ поменутих макроа. Иза њих је декларисана вредност којом се одређује трајање тачке. Сва остала времена су одређена у односу на ово време. На овај начин може се лако променити брзина емитовања поруке. Затим су декларисани типови осталих променљивих.

Порука је дефинисана као променљива типа string користећи наредбу char msg[] = "BISER-". Последњи знак у овој поруци није слово. Ово је намерно учињено да се покаже како програм функционише када се појави вредност израза која не постоји унутар лабела (бројеви написани иза речи case).

Унутар блока void setup појављује се наредба k = sizeof(msg); која позива функцијски програм sizeof за одређивање величине стринга, односно броја знакова који га чине. Овде треба истаћи да су позиције сваког карактера одређене почевши од 0 (први с лева) до 5 (последњи). С обзиром да укупно има 6 карактера, користећи дељење по модулу

```
j = i % k;
```

израчунава се позиција карактера који треба превести у морзеов код.

Програм наставља извршење блока наредби који одговара вредности добијеној обрадом израза (msg[j]). Тако ће се прво извшити све наредбе које следе након ознаке case 66: до наредбе brake. Након тога извршава се прва наредба иза блока који одговара структури типа

```
switch (i++).
```

У конкретном случају намерно је више пута изостављена наредба brake како би се продужило са извршавањем наредби у саставу блокова који припадају словима S, I, и E.

Следеће слово (I) преводи се у одговарајући морзеов код почевши од линије са ознаком case 73. Иза ње нема наредбе brake јер је неопходно генерисати још једну тачку. С обзиром да је она последња у низу, иза ње следи пауза која траје дупло дуже па је зато на овом месту примењен одговарајући макро са називом dotEnd.

У конкретном програму слово R се преводи у морзеов код унутар блока наредби иза линије case 82. Крај овог блока наредби саджи наредбу brake како би се обезбедио излазак из switch структуре. Као што је већ истакнуто, непостојање ове наредбе на овом месту изазвало би грешку у генерисању морзе кода јер би програм наставио да извршава наредбе у блоку који следи лабелу case 66.

6.4. Сажетак

Управљачке наредбе не врше никакву обраду већ само преносе ток управљања на неки део програма. Поред селекција и циклуса у ову групу наредби спадају и наредбе за реализацију скокова.

Селекције су управљачке структуре које омогућавају условно извршавање наредби.

Циклуси су управљачке структуре које омогућавају понављање наредбе или блока наредби. ARDUINO C подржава три типа циклуса: WHILE, FOR и DO. Циклус типа WHILE представља управљачку структуру са излазом на врху. Извршење наредбе/наредби остварује се све док се не оствари услов наведен на почетку циклуса. Ово значи да број понављања није строго дефинисан односно може да се мења у зависности од конкретних стања, на пример вредности неке променљиве, пре уласка у циклус типа WHILE. У случају циклуса типа FOR извршење наредбе/наредби, унутар тела циклуса, остварује се све док се не достигне гранична вреднст

дефинисана унутар услова за окончање циклуса. У случају DO циклуса наредба/наредбе се извршавају бар једном јер се услов за напуштање циклуса, за разлику од претходно поменутих типова, проверава тек на крају.

Селекција типа switch омогућава директно извршење наредбе или блока наредби у складу са вредношћу резултата који се добија као резултат израчунавања израза наведеног између заграда непосредно иза речи switch.

У случају честог понављања групе наредби погодно је користити макрое како би се поједносатило писање и добио прегледнији облик програма.

Питања

1. Наведите основне управљачке структуре?
2. Објасните како се дефинише if - else селекција у случају када се током одлучивања може остварити један од четири исхода.
3. У чему је разлика између WHILE, FOR и DO циклуса?
4. Шта су макрои и како се дефинишу?

7

АД И ДА КОНВЕРЗИЈА

У поглављу су описани суштина поступка аналогно-дигиталне конверзије који се користи у конкретном микроконтролеру. С обзиром да конкретан микроконтролер не садржи дигитално-аналогни конвертор описан је поступак апроиксимативног приступа у поступку добијања анлогног сигнала који се заснива на такозваној ширинско пулсној модулацији (PWM). Цео поступак је илустрован приказом одговарјућег примера.

7.1. Аналогно-дигитална конверзија

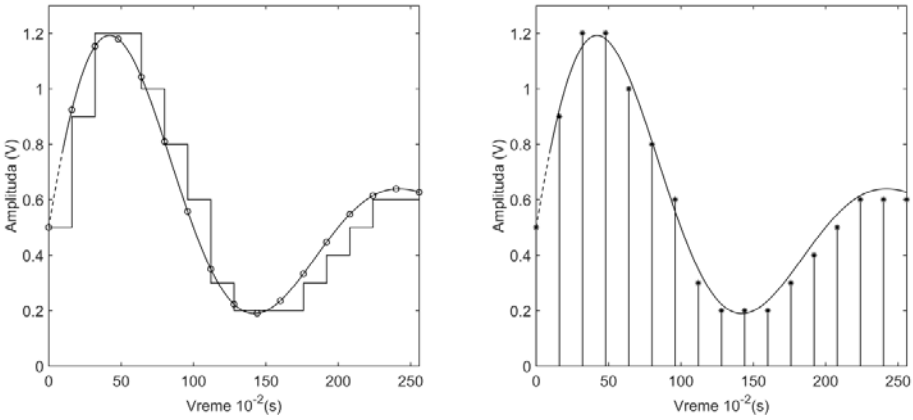
Процес превођења сигнала из аналогног у бинарни облик познат је под називом аналогно-дигитална конверзија и састоји се из два поступка: дискретизације (одабира) по времену и квантовања по нивоу.

Одабирање сигнала у времену најчешће се остварује униформно, односно у једнаким временским интервалима T_s. Овим поступком, аналогни сигнал f(t) замењује се поворком одбирака {f(0), f(T_s), f($2T_s$),...f(kT_s)}, који представљају његове вредности у тренуцима одабирања t = kT_s, (k = 0,1, 2,…). Регистроване вредности одбирака морају остати очуване до завршетка конверзије како би се остварио процес квантизације по нивоу, односно превођење у нумерички облик. Очигледно, процес дискретизације садржи две операције: одабирање (*sampling*) и задршку (*hold*).

Оба ова процеса, одабирање и квантовање по нивоу, у пракси се остварују користећи електронско коло познато под називом аналогно-дигитални конвертор (AD - *analog to digital conversion*). На излазу AD конвертора појављују се дигиталне речи са димензијом која је одређена величином његовог излазног регистра.

Цео процес дискретизације илустрован је на слици 7.1. На левој страни (а) приказан је аналогни сигнал (глатка линија) на којој су кружићима означене вредности које се одабирају у дискретним тренуцима времена за потребе аналогно-дигиталне конверзије. На излазу AD конвертора појављују се подаци у бинарном облику. Коначна дужина бинарне речи условљава постојање одступања у односу на аналогни сигнал у тренутку одабирања. Њихова вредност се усваја као његова апроксимација до следећег тренутка одабирања (степенаста линија на истој слици). Дискретне вредности, апроксимације аналогог сигнала, приказане су

вертикалним линијама са тачком на врху (десна страна слике, део (б)).



(а) (б)

Слика 7.1. Процес дискретизације; а) глатка непрекидна крива - аналогни сигнал; кружићи - његове вредности у тренутку одабирања; степенаста линија - његова дискретна апроксимација; б) одбирци сигнала након дискретизације по нивоу и времену.

AD конвертор, унутар микроконтролера који су уграђени у ARDUINO платформе типа UNO и NANO, генерише бинарне податке дужине десет бита. При томе се конверзија аналогних одбирака у бинарне податке заснива на поступку познатом под називом сукцесивна апроксимација. Објаснимо суштину овог поступка уз претпоставку да AD конвертор генерише бинарне речи дужине 4 бита. Ово значи да се поступак конверзије одвија кроз четири узастопна корака. Нека аналогни сигнал има вредност која одговара декадском броју 7 (бинарно 0111). Конверзија започиње тако што су иницијалне вредности свих бита нула.

Уколико се бит највише вредности постави у стање које одговара вишем логичком нивоу, односно имамо на његовом месту цифру 1, тада се добија бинарана реч 1000 која

одговара декадском броју 8. Како је ова вредност већа од 7 први бит мора остати на нивоу 0. Након тога поставља се први следећи бит нижег значаја на ниво 1. Новодобијена бинарна комбинација (0100) одговара декадској цифри 4 што је мање од 7. С обзиром да је добијена вредност мања од вредности која одговара аналогном сигналу на овој позицији остаје логичка јединица, а испитивање се наставља постављањем следећег бита нижег значаја на ниво 1. У овом кораку новодобијена бинарна комбинација (0110) одговара декадској цифри 6 што је мање од 7. Како се поново добија вредност мања од оне која одговара аналогном сигналу на овој позицији остаје логичка јединица, а испитивање се наставља постављањем бита најнижег значаја на ниво 1. Коначно се добија бинарна реч 0111 (декадска цифра 7) која одговара бинарном еквиваленту сигнала на улазу у AD конвертор. Уочимо да је за реализацију поступка сукцесивне апроксимације поред AD конвертора неопходно да одговарајући хардвер садржи дигитално-аналогни (DA) конвертор и компаратор аналогних сигнала. Ово је разумљиво када се има у виду да је улаз у систем аналогни сигнал. У сваком кораку поступка сукцесивне апроксимације неопходно је вратити тренутно добијени резултат, помоћу DA конвертора у аналогни облик јер га је једино тако могуће упоредити са вредношћу улазног сигнала који је у аналогном облику.

Тачност апроксимације улазног сигнала очигледно зависи од дужине излазног регистра AD конвертора, односно дужине бинарне речи која се појављује на његовом излазу. Што је дужина ове речи већа величина кванта по нивоу постаје мања. Другим речима постиже се већа резолуција што значи да бинарни податак прецизније описује стварну вредност аналогног сигнала на улазу AD конвертора.

ARDUINO платформе типа NANO располажу са 8 аналогних улаза (A0 – A7) што значи да постоји огућност

очитавања 8 аналогних сигнала. С обзиром да уграђени микроконтролер садржи само један AD конвертор очитавања се врше сукцесивно у различитим временским тренуцима. Очигледно, при дефинисању наредбе за AD конверзију биће неопходно да се наведе број улаза на којем се очитава вредност аналогног сигнала као и да се наведе назив променљиве којој се додељује ова вредност. На пример:

```
sensorValue = analogRead(A0);
```

При томе променљива треба да припада типу int како би се омогућио прихват десетобитног податка. У случају сукцесивног очитавања различитих аналогних улаза препоручује се убацивање наредбе за кашњење између наредби за AD конверзију да би се умањила могућност грешки у резултату које могу настати услед процеса преласка са једног на други аналогни улаз.

7.1.1. Очитавање напона на излазу интегратора

У примеру који следи (програм AnalogReadRC) описују се два процеса. Први од њих омогућава генерисање поворке правоугаоних импулса на дигиталном излазу са ознаком D2. Ови импулси представљају побудни сигнал за електрично коло састављено од само два пасивна елемента, отпорника R и кондензатора C. Они су повезани тако да формијрају интегратор (RC члан). При томе су крајеви отпорника R повезани на прикључке микроконтролера D2 и A0, а кондензатор C је спојен између прикључка A0 и масе.

Други процес омогућава мерење напона на излазу интегратора преко аналогног улаза A0. Мерење се остварује у једнаким временским интервалима, применом десетобитног AD конвертора који је саставни део микроконтролера. Програм AnalogReadRC садржи следеће линије кода:

```
/*AnalogReadRC*/
#define dpin 2
#define tSampl 5    // perioda odabiranja
int sensorValue;    // rezultat AD converzije
void setup() {
        Serial.begin(250000);
        pinMode(dpin, OUTPUT);
        digitalWrite(dpin, LOW);
        delay(2000);
}
void loop() {
        digitalWrite(dpin, HIGH);                    // turn out HIGH
        for (int j = 0, int n = 500; j <n; j++){
                sensorValue = analogRead(A0); // AD conv.
                Serial.println(sensorValue);
                delay(tSampl);
        }
        digitalWrite(dpin, LOW);                     // turn out LOW
        for (int j = 0; int n = 500; j <n; j++){
                sensorValue = analogRead(A0); // AD conv.
                Serial.println(sensorValue);
                delay(tSampl);
        }
}
```

На почетку програма дефинисан је дигитални излаз (dpin, у овом случају D2) који се користи за генерисање побудног сигнала, време одабирања (tSampl - интервал између два узастопна мерења) и променљива која означава мерење (sensorValue).

Блок наредби унутар void setup функције започиње дефинисањем брзине преноса сигнала према серијском монитору, односно серијском плотеру. Дигитални прикључак D2 поставља се у излазни мод рада (OUT) након чега се

његово стање дефинише као логичка нула (LOW) у трајању од 2000 милисекунди. На овај начин је обезбеђено да процес интеграције започне без заостале електростатичке енергије, односно са празним кондензатором. Другим речима, обезбеђено је да вредност почетних услова буде нула.

Трајање импулса као и однос сигнал-пауза контролишу се унутар void loop функције. При томе се најпре излаз D2 доводи у стање логичке јединице (HIGH). Трајање овог стања одређено је унутар циклуса типа FOR:

```
for (int j = 0; j <n; j++){
  sensorValue = analogRead(A0);          // AD conv.
  Serial.println(sensorValue);
  delay(tSampl);
}
```

Уочимо да се локална променљива j током сваког циклуса инкрементира за један. С обзиром на почетну вредност контролне променљиве j = 0 услов за излазак из циклуса (j < n) налаже да се циклус понавља n пута.

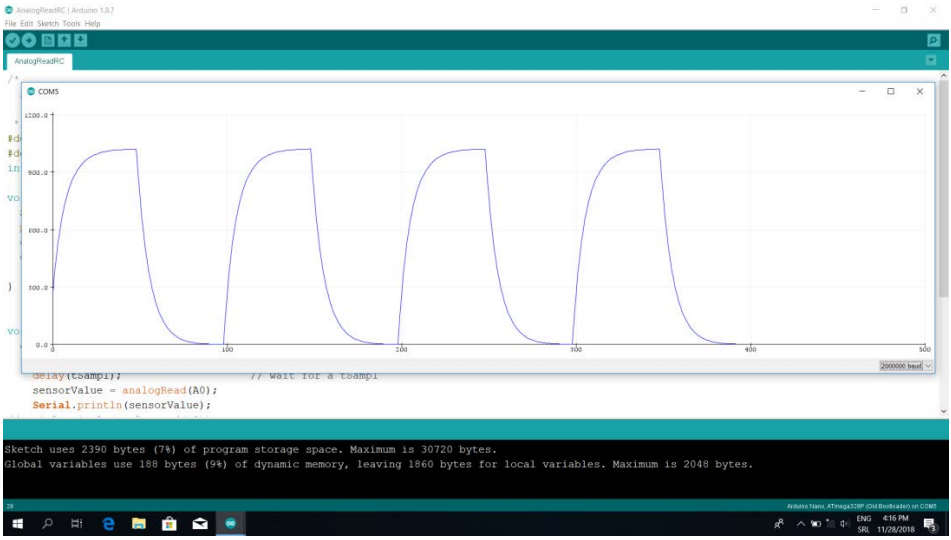
Прва наредба, унутар блока који следи, омогућава очитавање напона на улазу A0 и доделу одговарајуће бинарне вредности променљивој sensorValue. Ова вредност се прослеђује ка матичном рачунару како би се користећи ARDUINO програмско окружење, зависно од избора опције у менију Tools (Serial Monitor или Serial Plotter), омогућио приказ резултата на монитору рачунара (у виду нумеричких вредности или у графичком облику).

Уочимо да је трајање импулса одређено вредношћу умношка:

$$n \times tSampl = 500 \times 5 = 2500 \times 10^{-3}s,$$

односно 2500 милисекунди. С обзиром да су, у конкретном случају, вредности отпорника $R = 1\times10^{6}\ \Omega$ и кондензатора $C = 0{,}1\times10^{-6}$ F временска константа интегратора има вредност $T = R\times C = 0{,}1$ s, односно 100 милисекунди. С обзиром да је трајање прелазног процеса одређено интервалом дужине $5\times T$ (у случају строжијег критеријума $10\times T$) јасно је да ће дужина импулса бити сасвим довољна да се оконча прелазни процес, односно да напон на кондензатору поприми стационарну вредност која је једнака амплитуди побудног сигнала. Пелазни процес, у конкретном случају, има експоненцијални карактер.

По изласку из циклуса следи наредба којом се излаз D2 доводи у стање логичке нуле (LOW). Трајање овог стања одређено је, као и у претходном случају, унутар циклуса типа FOR. Пелазни процес, поново има експоненцијални карактер, али се напон мења од максималног ка нули. Након изласка из овог циклуса долази се до краја void loop блока па се цео процес понавља. Извршењем овог програма, на улазу AD конвертора настаје сигнал чија се промена може уочити на серијском плотеру у виду приказа који се види на слици 7.2.



Слика 7.2 Промена напона на излазу интегратора при побуди правоугаоним импулсима

Уколико се вредност променљиве n, у оба циклуса типа FOR, смањује за исти износ, прелазни процес све више поприма линеаран карактер. Ово је у складу са условом за добар рад интегратора (периода понављања сигнала мора бити значајно мања од временске константе интегратора). С обзиром да побудни сигнал, након промене, не мења вредност, при довољно малој периоди понављања излазни сигнал поприма облик нагибне функције (на пример за n = 2 периода сигнала је 20×10^{-3} s, односно 20 милисекунди) што је у складу са математичким поимањем интеграла (интеграл константе је нагибна функција).

Променом вредности локалних променљивих n унутар циклуса типа FOR може се мењати однос сигнал-пауза. У складу са природом прелазног процеса, за случај када однос сигнал-пауза није 1:1, при довољно малој периоди понављања у излазном сигналу ће се појавити истосмерна компонента чија вредност, порастом односа сигнал-пауза, постаје све већа.

7.2. Дигитално аналогна конверзија

Зависно од природе уређаја у окружењу ARDUINO платформе сигнал који се шаље треба да буде у дигиталном или аналогном облику. Аналогни сигнал се формира користећи специјално електронско коло познато под називом дигитално-аналогни (DA) конвертор. Међутим, у случају ARDUINO платформе типа NANO, уграђени микроконтролер не садржи дигитало-аналогни (DA) конвертор па је неопходно применити посебан поступак како би се створили услови за формирање аналогног сигнала. При томе се користи поступак познат под називом ширинско импулсна модулација (*Pulse Width Modulation* - PWM). Суштина овог поступка огледа се у генерисању поворке правоугаоних импулса са константном амплитудом и фреквенцијом понављања, али са могућношћу промене односа сигнал-пауза. Променом овог односа мења се средња вредност сигнала који се предаје потрошачу током сваке периоде. Уколико периоду понављања правоугаоног сигнала означимо словом *T*, а његову амплитуду и трајање импулса означимо словима *A* и *d* тада је средња вредност предатог сигнала

$$s_{out} = A\frac{d}{T}.$$

Програмска функција analogWrite омогућава генерисање правоугаоних импулса непроменљиве фреквенције са могућношћу задавања односа сигнал-пауза. Синтаксом ове функције:

analogWrite(analogOut,val)

захтева се навођење ознаке пина, односно прикључка (analogOut) на којем се генерише сигнал и вредности (val) у опсего од 0 до 255 којом се дефинише трајање импулса, а тиме и однос сигнал-пауза. У случају када је параметар val = 0 на конкретном прикључку се генерише непроменљиви сигнал

на нивоу LOW (0). Супротно томе када је val = 255 на овом излазу се генерише непроменљив сигнал на нивоу HIGH (1).

У примеру који следи описан је поступак генерисања трогаоног сигнала који се заснива на примени програмског PWM генератора (функција analogWrite).

На почетку програма, најпре је дефинисано да дигитални прикључак D9 буде излаз преко кога ће се слати PWM сигнал. Напоменимо да је избор прикључака преко којих се може генерисати PWM сигнал, за случај платформе типа NANO, ограничен на прикључке, D5, D6, D9, D10 и D11. Имајући у виду да је максимална могућа вредност трајања имулса (променљива val) одређена целим бројем 255 при дефинисању ове променљиве је усвојен тип byte.

```
/*PWM generator u funkciji generisanja signala zeljenog oblika */
#define analogOut 9                    // PWM out
int sensorValue ;                      // out of AD conv.
byte val = 0;                          // control variable
void setup() {
        Serial.begin(250000);
        // Inicijalizacija pocetnog napona (NULA)
        analogWrite(analogOut,val);
        delay(1000);
}
void loop() {
        for (val = 0; val <=64; val = val + 2){
                analogWrite(analogOut,val);          // PWM signal
                delay(60);                  // delay in between reads
                sensorValue = analogRead(A0);// read the in on analog pin 0:
                Serial.println(sensorValue/4);    // print out the value you read:
```

```
    }
    for (val = 64; val <=64; val = val - 2){
        analogWrite(analogOut,val);
        delay(60);
        sensorValue = analogRead(A0);
        Serial.println(sensorValue/4);
    }
}
```

Уочимо да је за дефинисање излаза analogOut, у приказаном програму, употребљена кључна реч #define. Унутар функције void setup, као и у претходном примеру, дефинисани су брзина преноса и почетни услови у RC колу (напон на крајевима кондензатора 0V). Унутар циклуса дефинисаног void loop функцијом уграђене су два циклуса типа FOR. Први од њих обезбеђује да се трајање импулса повећава у корацима који одговарају инкременту променљиве val (2) почевши од вредности 0 до вредности 64. Други циклус се реализује са истим кораком али у супротном смеру, односно смањујући трајање импулса од вредности 64 до вредности 0. Унутар оба циклуса уграђена је наредба која, након наредбе за генерисање импулса

```
analogWrite(analogOut,val);
```

обезбеђује кашњење у трајању од 60 милисекунди. Ово време треба да обезбеди успостављање одговарајућег напонског нивоа на крајевима кондензатора. С обзиром да се понављање импулса остварује сигналом чија је периода много краћа (у конкретном случају приближно 1 милисекунд) у односу на временску константу RC кола (100 милисекунди) напон на крајевима кондензатора лагано расте до нивоа који се региструје путем мерења заснованог на примени AD конвертора, користећи наредбу:

sensorValue = analogRead(A0).

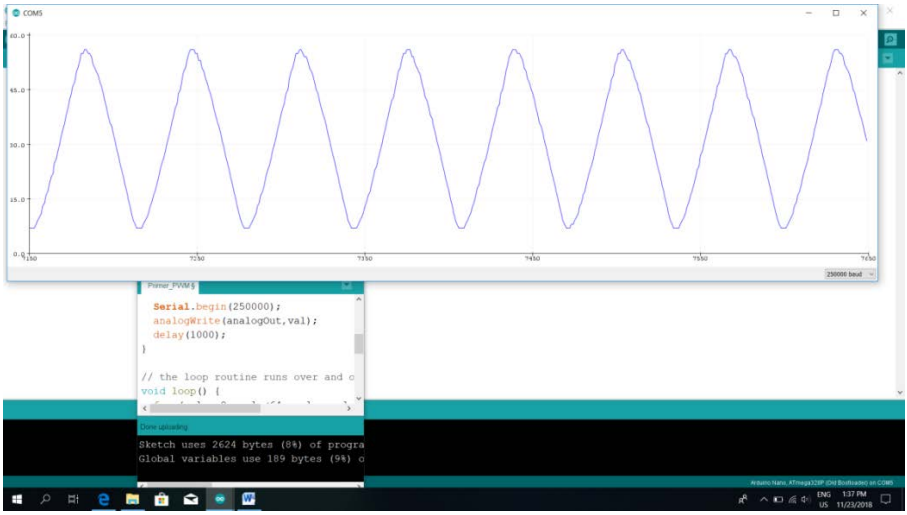
Након ове наредбе следи наредба за испис резултата на монитору рачунара

Serial.println(sensorValue/4)

у виду нумеричких вредности или у графичком облику, зависно од избора опције у менију Tools (Serial Monitor или Serial Plotter). Уочимо да је, у конкретном случају вредност променљиве на излазу AD конвертора подељена са 4 што одговара померању бинарног податка за два места у десно. Овим поступком десетобитни бинарни податак је преведен у осмобитни па га је могуће сместити унутар меморијске локације која одговара једном бајту. Уколико добијена резолуција одговара конкретним потребама, а постоји дефицит у погледу расположивог меморијског простора овај поступак представља добро решење за смештање великог броја мерених вредности.

Извршењем овог програма, на улазу AD конвертора настаје сигнал троугаоног облика. Његова промена се може уочити на серијском плотеру у виду приказа који се види на слици 7.3.

Очекује се да промена сигнала буде од нуле до неке максималне вредности, што не одговара приказу на слици. Уочавамо да је минималан ниво сигнала већи од нуле. Ова појава има везе са уграђеним кашњењем које је краће од временске константе RC кола. Прелазни процес током којег кондензатор треба да се у потпуности испразни не успева да се заврши до краја. Уколико се кашњење повећа на 300 милисекунди добија се одзив у којем сигнал мења своју вредност од нула до максимума и поново до нуле, током сваке периоде.



Слика 7.3 Сигнал на излазу интегратора при побуди PWM сигналом када ширина импулса прати промену амплитуде сигнала троугаоног облика.

Поред већ наведеног, продискутујмо услов изласка из циклуса типа FOR. Уочимо да се у оба случаја појављује исти услов (val <=64;). Ово је исправно када се има у виду природа промене променљиве типа byte. У случају када се вредност контролне променљиве (val) умањује, након вредности нула неће се појавити вредност са негативном предзнаком. Настаје прекорачење опсега дозвољених вредности за овај тип променљиве (дозвољено од 0 до 255) па ће поступак бити настављен у истом смеру тако да је прва наредна вредност 254, а не -2. Уколико би се унутар циклуса током којег се контролна променљива умањује услов дефинисао релацијским изразом (val >=0;) настала би грешка. Контролна променљива би након нуле попримила вредност 254, након чега би уследило умањење до нуле. Очигледно, након уласка у овај циклус извршење програма би се одвијало бесконачно унутар њега јер контролна променљива типа byte никада не поприма вредност која је мања од нуле. Овај услов би био прихватљив ако би контролна променљива припадала типу int (целобројне вредности).

7.3. Сажетак

Процес превођења сигнала из аналогног у бинарни облик познат је под називом аналогно-дигитална конверзија и састоји се из два поступка: дискретизације (одабира) по времену и квантовања по нивоу.

Тачност апроксимације улазног сигнала зависи од дужине излазног регистра AD конвертора, односно дужине бинарне речи која се појављује на његовом излазу. Што је дужина ове речи већа бинарни податак прецизније описује стварну вредност аналогног сигнала на улазу AD конвертора.

ARDUINO платформе типа NANO располажу са 8 аналогних улаза (A0 – A7) што значи да постоји огућност очитавања 8 аналогних сигнала.

Зависно од природе окружења ARDUINO платформе сигнал који се шаље треба да буде у дигиталном или аналогном облику. Аналогни сигнал се формира користећи дигитално-аналогни (DA) конвертор. У случају ARDUINO платформе типа NANO, уграђени микроконтролер не садржи дигитало-аналогни конвертор па је неопходно применити посебан поступак познат под називом ширинско пулсна модулација (PWM). Суштина овог поступка огледа се у генерисању поворке правоугаоних импулса са константном амплитудом и фреквенцијом понављања, али са могућношћу промене односа сигнал-пауза. Променом овог односа мења се средња вредност сигнала који се предаје потрошачу током сваке периоде.

Питања

1. Шта је АД конвертор?
2. Обајсните поступак примене ширинско пулсне модулације уместо ДА конвертора.

8

ПРЕНОС ПОДАТАКА

У поглављу су дефинисани основни поступци за асинхрони и синхрони пренос података. Дефинисани су појмови као што су бодска брзина и формат податка. Поред тога описани су и начини повезивања уређаја.

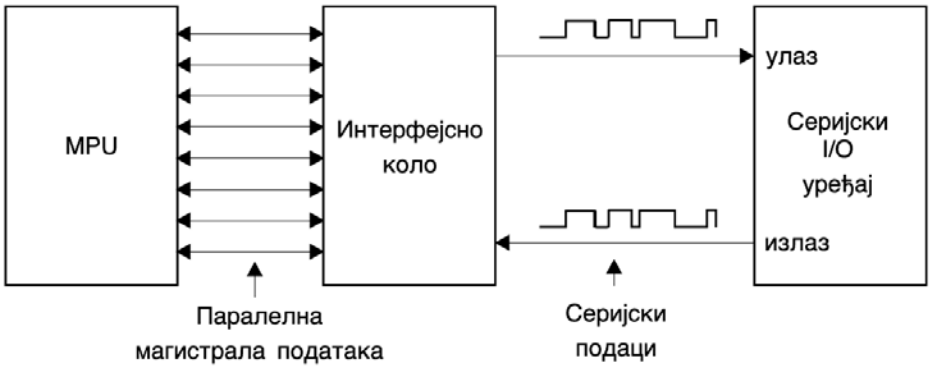
Размена података између микроконтролера и уређаја у његовом непосредном окружењу може да се оствари паралелним или серијским преносом. У случају паралелног преноса сваки бит податка се преноси истовремено па је неопходно постојање преносних водова у броју који одговара броју бита који се истовремено преносе. У случају серијског преноса, у сваком тренутку се преноси само један бит па је за пренос података могуће користити само једну жицу. Серијски пренос неминовно условљава постојање неког облика синхронизације да би се обезбедило правилно издвајање података из низа импулса који стижу на пријемну страну.

Уколико пренос података може да започне у било ком тренутку, када је преносна линија слободна, без присуства синхро сигнала, онда говоримо о асинхроном преносу. Наравно при томе је неизбежно постојање одговарајућих синхро импулса како би се означили крај и почетак податка.

У случају да се пренос сваког бита остварује пратећи промене додатног синхронизујућег сигнала онда говоримо о синхроном преносу. Током времена је развијено мноштво протокола за пренос података преко одговарајућих интерфејса. Наведимо само неке од њих: *Universal Serial Bus* – USB, *Serial Peripheral Interface* – SPI и *Inter Integrated Circuit*. – I^2C. У тексту који следи биће анализирани најчешћи облици серијског преноса. Најпре асинхрони, а затим синхрони.

8.1. Асинхрони пренос података

Када рачунар (у ствари било који уређај) треба да комуницира са серијским I/O уређајем, уобичајен начин је асинхрони серијски пренос. Слика 8.1 приказује блок дијаграм система за пренос података.



Слика 8.1 Интерфејс MPU-а и серијског уређаја

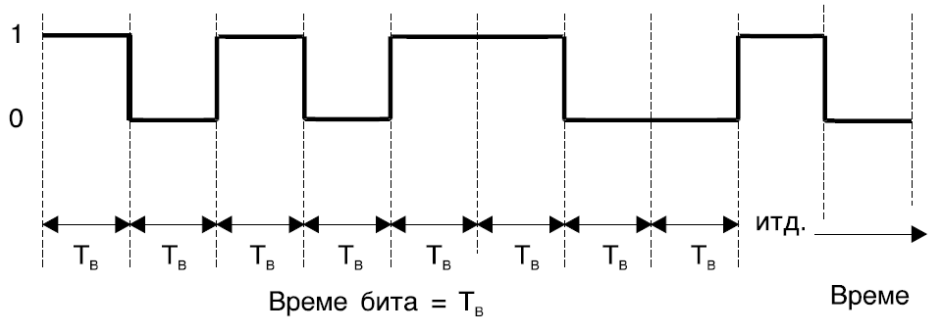
Систем извршава две основне операције:

- Узима 8-битну паралелну реч са MPU магистрале података и претвара је у серијску реч податка коју шаље ка серијском уређају.
- Узима серијски податак из серијског уређаја и претвара га у 8-битну паралелну реч податка коју шаље ка MPU-у преко магистрале података.

Очигледно, главна функција интерфејсног кола је претварање паралелних у серијске податке и обрнуто. Пре него што анализирамо како се ова конверзија може извршити, размотримо како изгледа сигнал који представља серијски формат податка.

8.1.1 Серијски формат података

Серијски сигнал податка је разложен у временске интервале који се називају трајање бита (битски интервал). За време сваког битског интервала (T_B) вредност сигнала може да буде 0 или 1. Сигнал може да мења нивое само на почетку сваког битског интервала.



Слика 8.2 Облик сигнала у случају серијског преноса

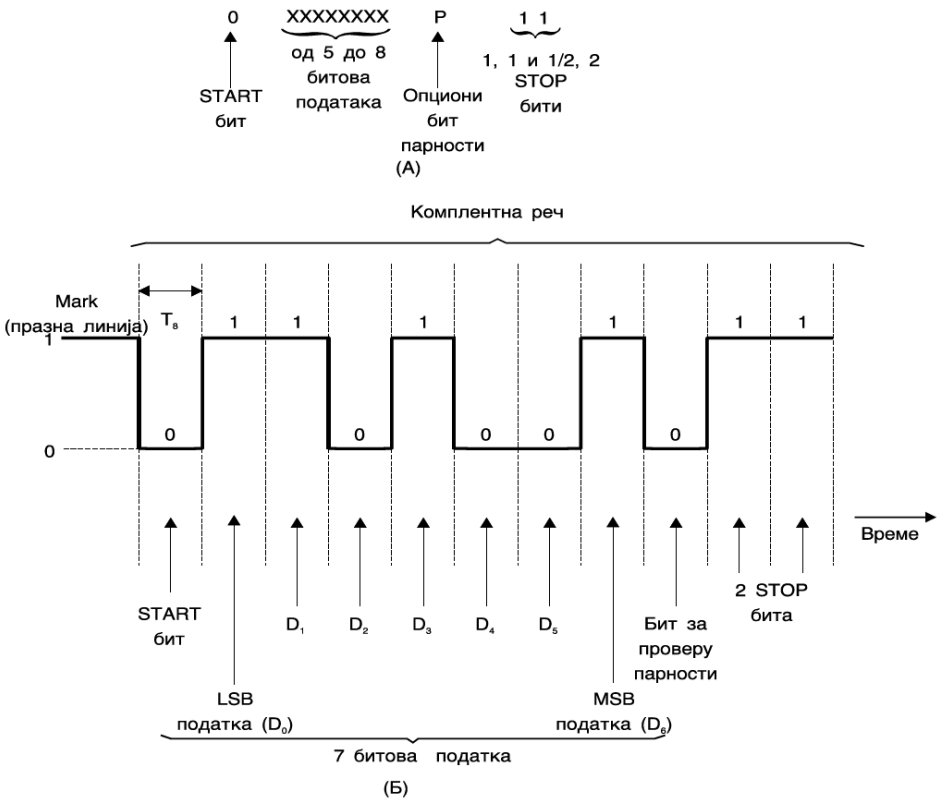
У случају асинхроног серијског преноса података користи се стандардни формат сигнала (сл. 8.3А) којег чине три (или опционо четири) дела:

1. START бит, који је увек 0
2. Пет до осам битова података, који представљају стварну информацију која се преноси.
3. Опциони бит парности који се користи за детекцију грешке. Уколико је бит парности укључен, може се користити парни или непарни број јединица.
4. Један, $1\frac{1}{2}$, или 2 STOP бита, који су увек 1.

За задати систем, број битова података, опција са битом парности, и број STOP битова зависе од пројектанта. На слици 8.3Б приказан је пример серијске речи која користи 7 битова података што је ASCII код за алфанумерички податак који се шаље.

Потпуни серијски податак на сл. 8.3Б почиње са START битом који је 0. Кад се на линију не шаље податак, на линији је висок напонски ниво (*idle state* - стање мировања). Почетак слања податка се означава преласком са 1 на 0, што је знак за појаву START бита. Иза START бита је 7 битова податка, који почињу са LSB и завршавају се са MSB. Тако је, стварни податак који је послат 1001011, што је ASCII код за карактер К. Иза битова података је бит парности, који је у нашем

примеру 0, пошто 7 битова садржи паран број јединица. Иза бита парности су 2 STOP бита, који су увек 1.

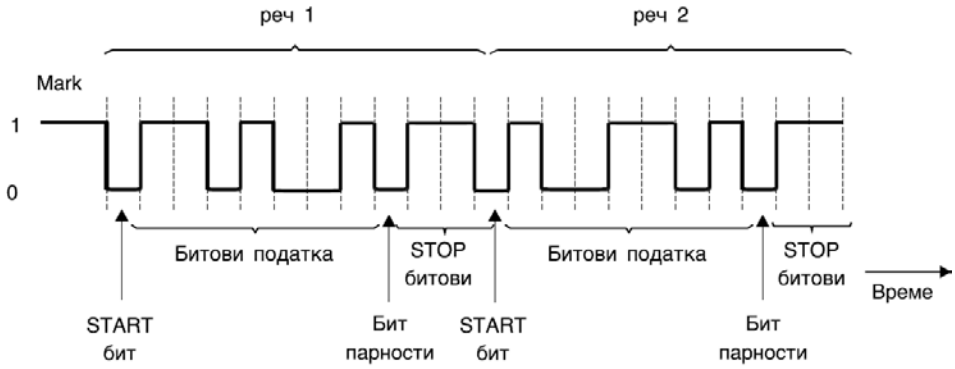


Слика 8.3 (А) стандардни асинхрони серијски формат података; (Б) пример серијске речи података која користи 7 битова података, бит парности (парни), и два 2 STOP бита.

Асинхрона серијска комуникација се обично користи за пренос неколико узастопних речи из једног у други уређај. Слика 8.4 показује серијски сигнал за пренос две узастопне речи. Прва реч је иста као на сл. 8.7Б. После 2 STOP бита прве речи, долази START бит друге речи иза кога су 7 битова, бит парности, и 2 STOP бита. Битови података друге речи су 10011001, што је ASCII код за Y. Када се серијски подаци шаљу један за другим, START бит сваке нове речи је одмах иза последњег STOP бита претходне речи. Овим је

представљена максимална брзина при преносу података. Када се серијски подаци не шаљу максималном брзином, предајник шаље константан висок ниво (*mark*) између STOP бита последње речи и START бита следеће речи.

Модул који прима серијски сигнал синхронизоваће се са негативном транзицијом коју формира START бит прве речи. На тај начин, модул „препознаје“ да су следећих 10 битова податак, парност и 2 STOP бита. После пријема STOP битова, пријемни уређај чека на следећу негативну транзицију, знајући да она представља START бит следеће речи. На овај начин су START и STOP битови употребљени да би синхронизовали серијски пријемник података на серијски предајник података.



Слика 8.4 Слање серијског сигнала за две узастопне речи

8.1.2. Бодска брзина

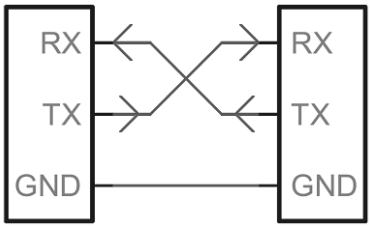
Брзина којом се одвија серијска предаја назива се бодска брзина (*baud rate*). Она је једнака броју битова информација које се шаљу у секунди. Пошто се 1 бит шаље у временском интервалу једнаком битском интервалу T_B, бодска брзина је дата са:

$$\text{бодска брзина} = \frac{1}{T_B} \text{ bit/s}$$

На пример, у системима са телепринтерима, T_B је 8.09 милисекунде. Ово је бодска брзина од 110 bit/s. За видео дисплеј терминал, типична бодска брзина је 19 200 bit/s. За правилно тумачење примљених података, неопходно је да пријемник серијских података ради истом бодском брзином као предајник.

8.1.3. Повезивање уређаја

Уређаји који користе серијски пренос података имају два прикључка: RX за пријем и TX за слање. Повезивање два уређаја остварује се унакрсним спајањем ових прикључака.



Слика 8.5 Повезивање уређаја у случају асинхроне серијске комуникације.

Предајни прикључак једног уређаја повезан је са пријемним прикључком другог и обрнуто. Постојање двоструке (*duplex*) везе омогућава истовремено слање поруке у оба смера. Ово је такозвани *full-duplex* мод рада. Уколико се комуникација одвија тако што пријемни уређај одговара тек након што прими поруку од предајног уређаја онда се каже да уређаји раде у *half-duplex* моду. У том случају могуће је користити само једну физичку везу, односно један вод за пренос података. При томе, на свакој страни мора постојати одговарајући блок електронике који обезбеђује наизменично укључивање предајног и пријемног подсистема. Након сваке поруке, на преносном воду се успоставља неактивно стање (*idle state*) па су оба уређаја на пријему. Уколико један од њих треба да упути поруку, тада се најпре проверава да ли је

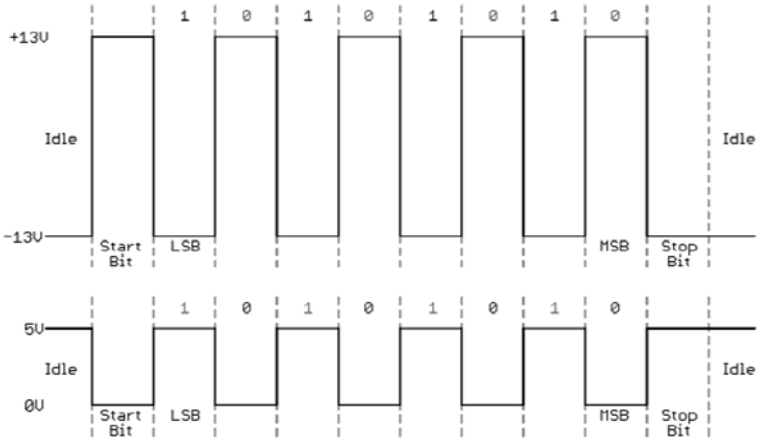
линија слободна (нема преноса) и у зависности од тога прелази на предају.

Наравно постоје ситуације када је један од уређаја увек на пријему. Пример такве везе представља LCD монитор чији је RX прикључак спојен са TX прикључком ARDUINO платформе.

Постоји низ стандарда који дефинишу облик сигнала при серијском преносу података. Размотримо само два који се чешће срећу у пракси а односе се на размену података у окружењу TTL (*Transistor-Transistor Logic*) интегрисаних кола и RS-232 стандарад.

У случају серијске комуникације микроконтролера са интегрисаним колима нижег нивоа обично се користе напони од 0V до 3,3V или 5V. Сигнал на нивоу напона напајања (3,3V, 5V) означава стање мировања (*idle state*), вредност бита која одговара логичкој јединици или STOP бит. Напонски ниво 0V који одговара маси означава START бит или вредност бита која одговара логичкој нули.

Стандарад означен као RS-232 подразумева обликовање сигнала који заправо представљају инверзну слику онога што се појављује код ТТЛ кола. Сигнали могу мењати вредност у опсегу од +/-3V до +/- 25V. У примеру на слици која следи нисконапонски сигнал (-13V) означава стање мировања, вредност бита која одговара логичкој јединици или STOP бит. Виши напонски ниво (13V) означава START бит или вредност бита која одговара логичкој нули. На слици 8.6 дат је упоредни приказ сигнала за оба случаја (TTL кола и RS-232 стандард).



Слика 8.6 Временски дијаграм сигнала који преноси бинарни код 01010101; у случају RS-232 стандарда (горе) и за TTL кола (доле).

Овај стандард се користи при преносу података на већа растојања због веће отпорности на сметње које могу настати у окружењу. Поред њега, за пренос података на већа растојања користи се и нешто комплекснији стандард познат под називом RS-485.

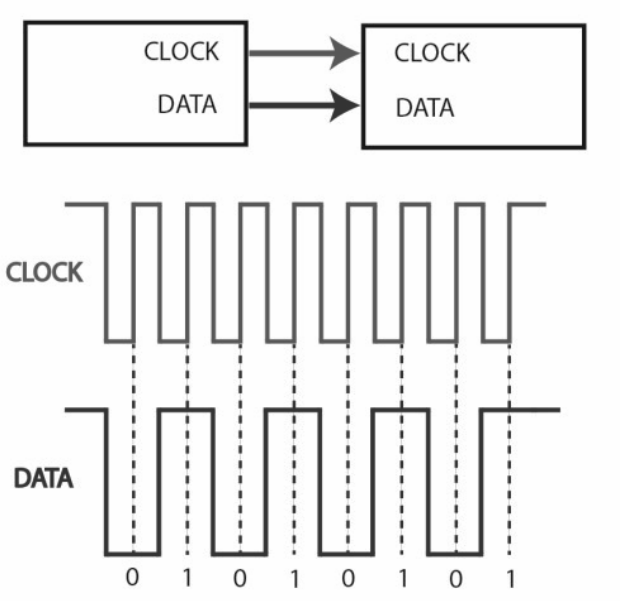
8.2. Синхрони серијски пренос података

Синхрони серијски пренос података типа SPI (**Serial Peripheral Interface**) представља уобичајени тип комуникацијског протокола који се користи између микроконтролера и уређаја у непосредном окружењу (на пример LCD монитор или меморијска картица). Бржи је у односу на асинхрони пренос и I^2C пренос.

8.2.1. Основе SPI преноса

SPI је синхрони тип протокола јер се пренос сваког бита остварује у строго дефинисаним временским тренуцима, односно у складу са променама синхронизујућег сигнала CLK (Clock signal). Брзина преноса је дефинисана динамиком

синхронизујућег сигнала. При томе треба имати у виду ограничења која проистичу из фреквенцијских карактеристика преносних водова и самих примопредајних интерфејса. Слика 8.7 приказује најпростији облик SPI комуникације.

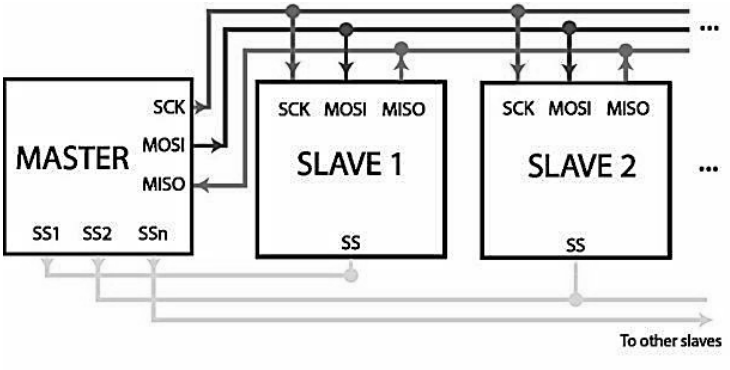


Слика 8.7 Најпростији облик повезивања уређаја у случају SPI протокола (горе) и временски дијаграм одговарајућих сигнала (доле).

Главни (мастер) уређај емитује синхронизујући сигнал (CLK) и податке (DATA) ка споредном (слејв) уређају. У конкретном случају очитавање сваког бита се остварује у тренуцима када се вредност CLK сигнала мења са нижег на виши ниво (растућа ивица). На овај начин је обезбеђена једносмерна комуникација и то од уређаја који генерише CLK сигнал (Master) ка пријемном уређају (Slave).

За двосмерну комуникацију мора постојати још један вод којим се подаци могу преносити од Slave уређаја ка Master уређају. Уколико постоји више пријемних уређаја неопходно је увести и четврти вод како би се омогућио одабир

„саговорника“. Слика 8.8 приказује организацију физичких веза у случају када се комуникација остварује са два пријемна уређаја.



Слика 8.8 SPI комуникација у случају паралелне везе више пријемних уређаја.

Очигледно, за реализацију SPI протокола уређаји морају да имају најмање три прикључка **MOSI (Master-Out-Slave-In), MISO (Master-In-Slave-Out) и CLK**. Ово указује на чињеницу да SPI функционише по принципу *master-slave* организације (један уређај управља, а други само прати његове захтеве) и користи *full-duplex* комуникацију (истовремено је могућ пренос у оба смера). Наравно, у случају више пријемних уређаја мора постојати и прикључак SS (Slave Select) како би се омогућио одабир уређаја са којим се тренутно комуницира.

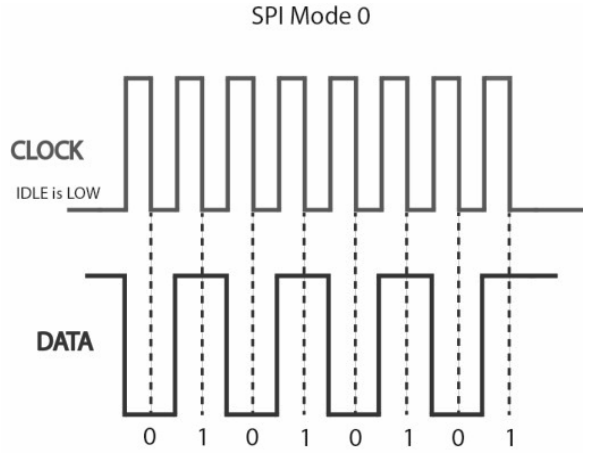
Мастер уређај одређује параметре SPI комуникације укључујући битску брзину, редослед бита и мод рада. Битска брзина је условљена фреквенцијом „такта“, односно сигнала који синхронизује све операције микроконтролера. Редослед бита описује који бит се прво шаље (крајњи десни LSB или крајњи леви MSB). Могућа су четири мода рада. Њихов детаљан опис дат је у табели:

Табела 8.1

Mode	Clock Polarity	Clock Phase	Clock is Idle at...	Data is Shifted Out During...
Mode 0	0	0	Low	Falling Edge
Mode 1	0	1	Low	Rising Edge
Mode 2	1	0	High	Rising Edge
Mode 3	1	1	High	Falling Edge

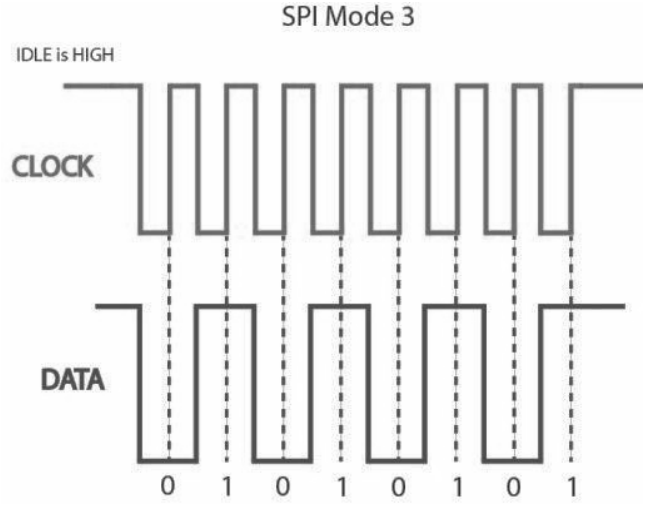
При томе clock polarity означава стање CLK сигнала када нема преноса док clock phase означава који део CLK импулса се користи (растући или опадајући) при активирању преноса конкретног бита.

Слика 8.9 приказује временски дијаграм сигнала за случај мода 0. У овом моду, пренос сваког бита започиње у тренутку појаве силазне ивице CLK сигнала. У стању мировања овај сигнал је на нивоу који одговара логичкој нули (LOW). Након слања сваког бита импулси CLK сигнала прелазе у стање (HIGH).



Слика 8.9 Временски дијаграм сигнала у моду 0.

Ради поређења на следећој слици је приказан случај када се пренос сваког бита остварује наиласком растуће ивице CLK сигнала, а стање мировања означава вредност овог сигнала која одговара вишем логичком ниво (HIGH). Након слања сваког бита импулси CLK сигнала прелазе у стање (LOW). Овај начин рада представља мод 3.



Слика 8.10 Временски дијаграм сигнала у моду 3.

8.2.2. *Основе I^2C (Inter Integrated Circuit) преноса*

Овај тип протокола је развијен за пренос података између уређаја који се налазе на врло блиским растојањима, односно за размену података између интегрисаних кола унутар неког уређаја (single board communication). На пример, као у случају преноса података између микроконтролера и LCD монитора. За реализацију преноса података, у случају овог протокола, неопходне су две жице па се у литератури често среће под називом „*2 wire protocol*”. Због веома малих растојања пренос података може да се остварује брзином од неколико MHz. При томе једна жица омогућава пренос

података (SDA), а друга обезбеђује пренос синхронизирајућег сигнала (SCL).

Структура I^2C протокола је дефинисана тако да се након генерисања START сигнала најпре шаље адреса уређаја (ADR) са којим се успоставља комуникација. Адресу чини садржај седам узастопних бита након којих следи осми бит за дефинисање смера комуникације (READ/WRITE). У случају када главни уређај (*master*) шаље податак према неком од умрежених уређаја (*slaves*) овај бит се дефинише као ниво који одговара логичкој нули (LOW). У супротном случају, односно када главни уређај захтева да му неки од умрежених уређаја пошаље податак овај бит се дефинише као ниво који одговара логичкој јединици (HIGH). Након слања осмог бита главни уређај чека на одговор уређаја чија адреса одговара адреси послатој унутар првих седам бита. У случају подударања ових адреса конкретни пријемни уређај (slave) шаље бит потврде (acknowledge bit ACK) након чега главни уређај шаље осмобитни податак. По пријему податка пријемни уређај (slave) поново шаље бит потврде ACK. Описани процес се понавља све док главни уређај не најави прекид комуникације тако што изгенерише STOP сигнал, након пријема последњег бита потврде. Структура I^2C протокола, у случају када главни уређај шаље податак, детаљније је приказана на слици 8.11. Очигледно, пријемни уређај шаље само сигнал потврде пријема ACK.

START	ADRESA	WRITE	ACK	DATA	ACK	. . . . .	DATA	ACK	STOP

Слика 8.11 Приказ структуре I^2C протокола када главни уређај шаље податак. Сигнал који шаље пријемни уређај је приказан осенченим пољем.

У случају када главни уређај треба да прими падатке од неког од умрежених уређаја одвија се сличан процес. При томе се, након слања адресе уређаја са којим се успоставља комуникација, захтев за пријем података најављује тако што

осми бит садржи логичку јединицу, односно информацију READ. Након тога адресирани умрежени уређај (*slave*) шаље бит потврде ACK и наставља са слањем осмобитног податка. С обзиром да сада главни уређај прима податке, у наставку комуникације он потврђује пријем, односно генерише бит потврде пријема ACK. Прекид комуникације, главни уређај најављује тако што након пријема последњег податка уместо ACK шаље *not-acknowledge* NACK бит и затим генерише STOP сигнал.

Структура I^2C протокола, у случају када главни уређај прима податак, детаљније је приказана на слици 8.12. Очигледно, пријемни уређај шаље сигнал потврде пријема ACK само пре слања првог податка.

Слика 8.12 Приказ структуре I^2C протокола када главни уређај прима податке. Сигнал који шаље пријемни уређај приказан је осенченим пољем.

8.3. Сажетак

Размена података између микроконтролера и уређаја у његовом непосредном окружењу може да се оствари паралелним или серијским преносом. У случају серијског преноса, у сваком тренутку се преноси само један бит па је неопходно постојање неког облика синхронизације да би се обезбедило правилно издвајање података из низа импулса који стижу на пријемну страну. Уколико пренос података може да започне у било ком тренутку, онда говоримо о асинхроном преносу. Брзина преноса података је одређена бројем битова који се шаљу у секунди.

Синхрони пренос података се остварује у строго дефинисаним временским тренуцима, односно у складу са променама синхронизујућег сигнала CLK (*Clock signal*). Брзина преноса је дефинисана динамиком синхронизујућег сигнала.

При томе постоје ограничења која проистичу из фреквенцијских карактеристика преносних водова и самих примопредајних интерфејса.

Питања

1. Објасните суштину асинхроног и синхроног преноса података.
2. Како изгледа формат података у случају серијског преноса?
3. Како се остварује серијски пренос података у случају I^2C протокола?
4. Шта је битска брзина?
5. Објасните суштину RS-232 стандарда за пренос података.

9

ФУНКЦИЈЕ

У поглављу је приказано како се развија функција. Дефинисани су појмови као што су спецификација и тип функције, аргументи, тело функције и њен потпис.

Општа структура функције у језику C има форму у којој се најпре појављује њен наслов, а затим тело функције, односно блок наредби који се извршава при њеном позивању као у примеру који следи:

```
int NearestRound(float x) {
        // telo funkcije (naredbe):
}
```

Наслов функције започиње спецификацијом типа излазне променљиве након чега следи њен назив (у овом примеру NearestRound) и опис аргумената. У конкретном примеру захтева се да функција одреди најближу целобројну вредност која одговара променљивој x типа float. Функција треба да буде декларисана изван било које друге функције, пре или иза "loop()" функције.

9.1. Спецификација типа функције

Тип функције се наводи на почетку, односно непосредно пре исписивања блока наредби које она садржи. У нашем примеру, потребно је да функција врати најближу целобројну вредност. Спецификација типа функције треба да дефинише врсту података који се враћају када се функција позива.

Подаци који се враћају из функције могу припадати било којем типу (на пример double, long, char, byte,..). Ако функција не враћа вредност, онда се при спецификацији типа мора користити кључна реч void (као у случају void setup() и void loop()).

9.2. Име функције

Други део функције чини њен назив. При именовању функција важе иста правила која се користите за имена променљивих. Већина библиотечких функција почиње са

малим словом, иако то није обавезно. Уколико је важно да лако уочите имена функција које сте ви написали потребно је да примените неки специфичан начин означавања (на пример, да назив започне великим словом). Избор имена функције је битан. Добро име функције говори о томе шта она ради, али не и нужно како то ради. У конкретном примеру назив NearestRound јасно указује да функција одређује најближу целобројну вредност.

9.3. Аргументи функције

Након имена функције треба да се наведу њени аргументи. Њихово навођење започиње заградом након имена функције. Аргументи функције се користе за пренос података у функцију како би се остварио неки посебан поступак њихове обраде. Навођење више аргумената остварује се уз обавезно писање запета између њих. Често ћете чути да програмери уместо назива „аргументи функције" користе назив „листа аргумената". Ова листа се обавезно завршава затвореном заградом.

Гледано из угла корисника, функција може да се замисли као црна кутија са улазом и излазом. Корисник једино мора да зна које податке и којим редом треба да наведе како би се на излазу добио неопходан резултат. Сам поступак обраде може бити потпуно сакривен. На пример, у случају библиотечке функције Serial.print, знамо да се она користи за пренос података ка монитору, али при томе није видљиво како се то обавља. Морамо само знати како да правилно наведемо податке који се преносе.

Тип податка који се добија као резултат обраде унутар функције одговара типу који је наведен при њеној спецификацији. Уколико се извршењем функције не генерише резултат у облику податка онда се при њеној спецификацији обавезно наводи кључна реч void. У случају да се као резултат

добија податак типа float тада се при спецификацији функције обавезно мора навести тип double или float.

9.4. Тело функције

Тело функције започиње отвореном витичастом заградом "{", после навођења њених аргумената, а завршава затвореном витичастом заградом "}". Све наредбе између ова два знака представљају тело функције. Уколико је спецификатор типа функције била шта осим void, онда барем једна од наредби у телу функције мора садржати кључну реч return. На пример:

```
Int NearestRound(float x) {
        int c;
        c = (int)x;
        if ((x - c) > 0.5)
        c++;
        return c;
}
```

У овом примеру, при спецификацији функције, наведен је тип int па њено тело мора садржати наредбу return којом се враћа податак истог типа, односно int. У случају да се изостави наредба

```
return c;
```

у извештају би се, током компајлирања, појавила порука са описом грешке.

9.5. Потпис функције

Понекад, током дискусије о функцијама, можете чути термин потпис функције. Потпис функције се састоји од свега што следи након спецификације њеног типа до заграде на

крају листе аргумената. На пример за функцију NearestRound() потпис функције чини текст:

```
NearestRound(float x)
```

Другим речима, потпис функције нам указује на њено име и податке које треба пренети у њу као аргументе. Потписи функције су важни јер можете имати више од једне функције са истим именом. На пример, web страница (http://arduino.cc/en/Reference/Random) садржи опис функције random() на два начина:

```
random(max)
random(min, max)
```

Ако две функције имају исто име, како компајлер „зна“ коју треба да користи? Компајлер нема проблема да донесе ту одлуку јер су потписи функције различити. Уколико смо позвали функцију

```
random(min, max)
```

компајлер „зна“ да је потребно користити дефиницију функције која има листу аргумената са два аргумента. Кад год функција има два или више различита потписа, назива се преоптерећена функција (Overloaded Function). Често се користе два потписа како би се, у конкретној ситуацији, искористила она која лакше решава тренутни задатак. На пример, функција

```
random(300)
```

омогућава генерисање случајног броја у опсегу од 0 до 299. Међутим функција

```
random(10, 20)
```

омогућава да се генерише случајан број у опсегу од 10 до 19.

9.6. Сажетак

Општа структура функције у језику C има форму у којој се најпре појављује њен наслов, а затим назив и опис аргумената. Функција треба да буде декларисана изван било које друге функције, пре или иза "loop()" функције. Аргументи функције се користе за пренос података у функцију како би се остварио неки посебан поступак њихове обраде. Навођење више аргумената остварује се уз обавезно писање запета између њих. Уколико се извршењем функције не генерише резултат у облику податка онда се при њеној спецификацији обавезно наводи кључна реч void.

Тело функције започиње отвореном витичастом заградом ("{"), после навођења њених аргумената, а завршава затвореном витичастом заградом("}"). Све наредбе између ова два знака представљају тело функције.

Уколико је, при спецификацији функције, наведен тип функције њено тело мора садржати наредбу return којом се враћа податак истог типа. Потпис функције се састоји од свега што следи након спецификације њеног типа до заграде на крају листе аргумената. Другим речима, потпис функције нам указује на њено име и податке које треба пренети у њу као аргументе. Потписи функције су важни јер можете имати више од једне функције са истим именом. Ако две функције имају исто име, компајлер зна коју треба да користи јер су потписи функције различити.

Питања

1. Како се декларише функција?
2. Шта су тип и аргументи функције?
3. Шта, у телу функције, означава кључна реч return?
4. Шта је потпис функције?
5. Ако две функције имају исто име, како компајлер зна коју треба да користи?

10

ПРЕКИДИ

У поглављу је дефинисана употреба прекида. Описан је поступак обраде спољног прекида. Поред тога дат је пример којим се илуструје развој функцијског програма за обраду прекида.

Из до садашњег текста упознали смо основне принципе рада микроконтролера. Целокупан процес се одвија реализацијом наредби по принципу одозго на доле осим у случају када се наиђе на цикличну управљачку структуру (FOR, WHILE и DO). У случају ARDUINO платформи главни програм се извршава кроз бескрајно понављање секвенце наредби унутар функције loop(). Међутим, понекад је неопходно променити ток програма због настанка неког догађаја у окружењу микроконтролера. Овај тип активности је познат под називом прекид и условљен је променом стања на неком унапред дефинисаном прикључку.

10.1. Употреба прекида

Употреба прекида захтева увођење посебних програмских целина, односно функцијских потпрограма који се најчешће називају сервисна рутина за обраду прекида (*Interrupt Service Routine* - ISR). Уколико конкретан потпрограм за обраду прекида назовемо isr() тада одговарајући програм микроконтролера обавезно садржи следеће функције:

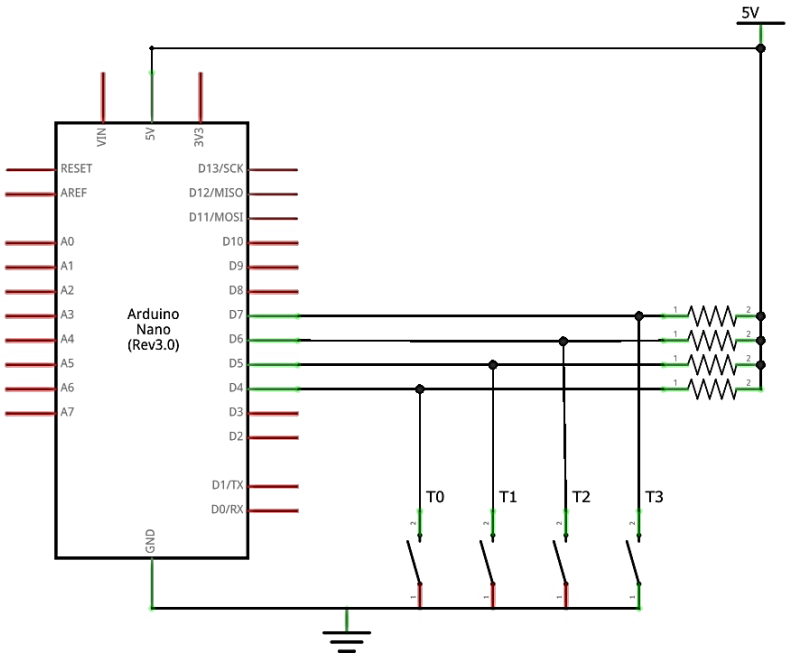
```
void setup(){
}

void loop(){
}

void isr(){
}
```

Улоге функција setup() и loop() већ су довољно коментарисане. У моменту настанка прекида ток програма ће се преусмерити на извршење функције isr(). Приметимо да је у њеном дефинисању наведена реч void. Као што је познато ово указује да се њеним извршењем не очекује враћање вредности неке променљиве у главни програм.

Оптимално коришћење ресурса неког микроконтролера могуће је само уколико га ослободимо од послова као што је непрекидна провера појаве промена на појединим прикључцима. Претпоставимо да у непосредном окружењу микроконтролера постоје четири тастера преко којих се може мењати ток програма. Као што се види на слици 10.1 сваки од њих је повезан између одговарајућих прикључака микроконтролера и масе.



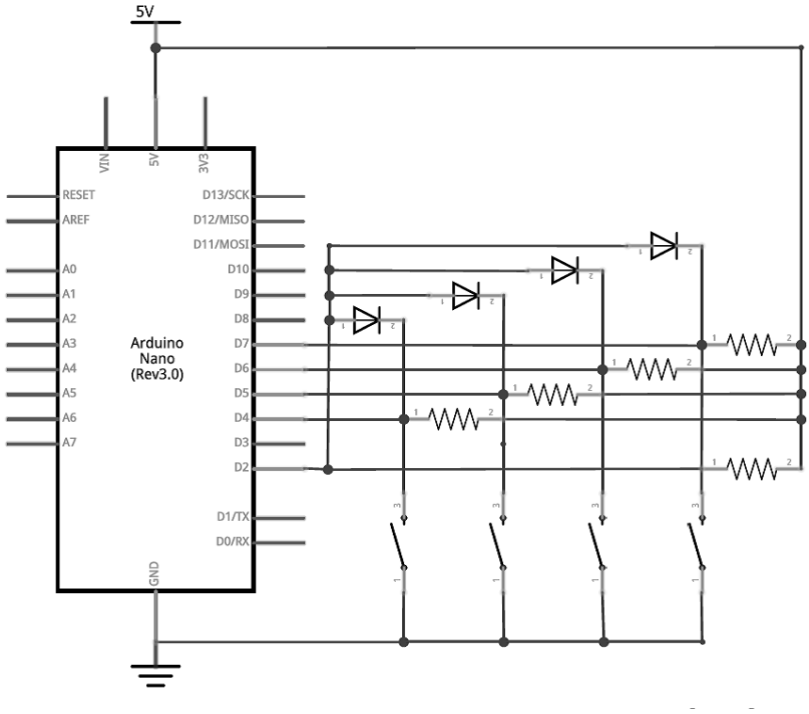
Слика 10.1 Принципијелна шема веза за контролу програма преко спољашњих тастера

При томе је сваки прикључак повезан са напајањем преко *pull-up* отпорника како би се поставио у више логичко стање (HIGH) када су тастери отворени.

Уочимо да постојање наведених отпорника није неопходно у случају микроконтролера ATMEGA328p јер се исто може постићи применом *pull-up* отпорника који већ

постоје унутар микроконтролера. Из едукативних разлога, ова веза је ипак наглашена додатком спољних отпорника у шеми приказаној на слици 10.1. Притиском на тастер одговарајући прикључак се спаја на масу и доводи у стање логичке нуле (LOW). У делу програма који се непрекидно понавља (void loop) проверава се да ли је неки од тастера притиснут. У зависности од резултата извршиће се одговарајућа секвенца програма.

Уколико се шема веза незнатно прошири, као на слици 10.2, микроконтролер не мора више да проверава стање прекидача током сукцесивног извршења наредби унутар циклуса (void loop). Новом везом, сваки пут када се притисне неки од тастера настаће и промена стања на D2 прикључку развојне платформе NANO. Једна од карактеристика овог прикључка је управо могућност прихватања спољног прекида у циљу преусмеравања тренутног тока програма. При томе се, у наменски дефинисаном простору интерне меморије микроконтролера, најпре меморишу вредности тренутних садржаја регистара као и адреса меморијске локације на којој се налази инструкција коју треба извршити након повратка у главни програм. Процес се наставља тако што се у програмски бројач уписује адреса подпрограма за обраду конкретног прекида. Листа адреса за обраду захтева који је условљен настанком прекида смештена је, у складу са њиховим приоритетима, на самом почетку меморијског простора.



Слика 10.2 Принципијелна шема веза за контролу тока програма применом спољних прекида

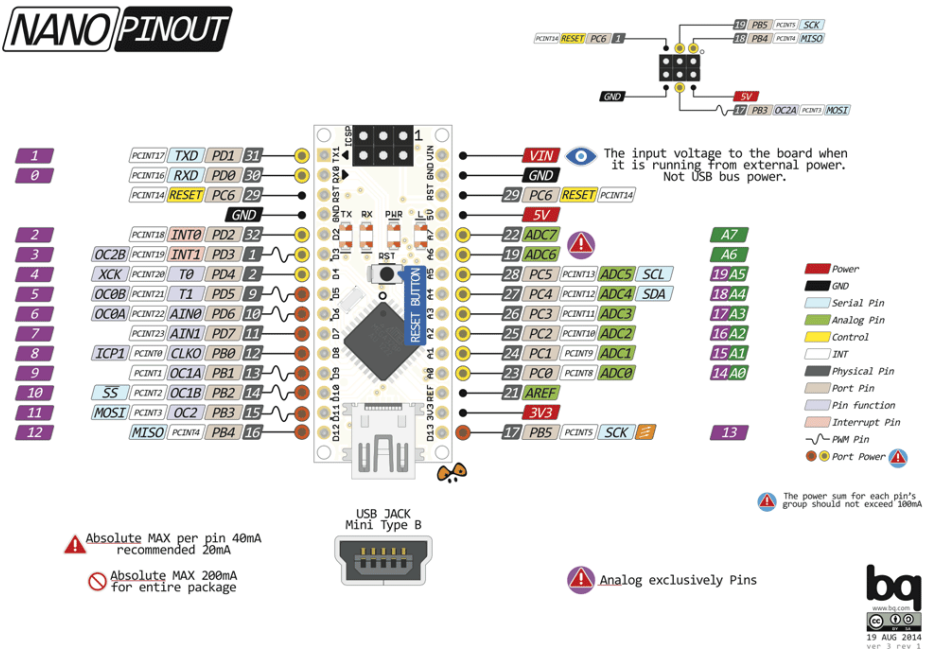
Слика 10.3 приказује листу могућих прекида за микроконтролер ATMEGA328p укључујући њихове називе и двобајтне адресе које ће бити унете у програмски бројач ради преумеравања тока програма на скуп инструкција за обраду конкретног прекида.

Уочимо да меморијска локација са адресом 0x0000 садржи информацију о адреси прекида са највишим приоритетом RESET. Ово је заправо место на којем се налази адреса меморијске локације од које треба да започне извршење програма сваки пут када укључимо микроконтролер или намерно изазовемо RESET, у циљу успоставе почетног стања, односно реиницијализације (такозвано ресетовање).

Vector No	Program Address(2)	Source	Interrupts definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Coutner2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Coutner0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART Data Register Empty
21	0x0028	USART_TX	USART Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator

Слика 10.3 Листа могућих прекида за микроконтролер ATMEGA328p.

Након завршетка потпрограма за обраду прекида у програмски бројач се уписује адреса следеће инструкције коју треба извршити у главном програму. При томе се, најпре, у одговарајућим регистрима поново успостављају вредности променљивих које су коришћене у главном програму пре настанка прекида. Слика 10.4 даје увид у могућности коришћења сваког од прикључака на платформи NANO.



Слика 10.4 Приказ могућности коришћења сваког од прикључака на платформи NANO.

Развојна платформа типа NANO нуди могућност коришћења још једног прекида на прикључку D3. Његова ознака је INT1 са адресом која се налази на локацији 0x0004

10.2. Дефинисање обраде спољног прекида

Прекиди типа INT0 и INT1 су могући само уколико се унутар (void setup) функције уведу наредбе за позив одговарајућег функцијског програма, у конкретном случају, attachInterrupt којим се активност микроконтролера преусмерава на обраду конкретног прекида. У тексту који следи наведен је пример програма који омогућава примену спољашњег прекида за контролу рада светлеће диоде (LED):

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
```

```
void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(interruptPin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}
void loop() {
    digitalWrite(ledPin, state);
}
void blink() {
    state = !state;
}
```

На почетку програма најпре су дефинисане глобалне променљиве: ledPin (име придодато светлећој диоди која је уграђена на NANO платформи и повезана са прикључком D13 као излазни уређај), interruptPin (прикључак D2 који се користи за регистровање спољшњег прекида) и state (променљива која се појављује унутар потпрограма за обраду прекида).

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
```

У оквиру функције setup() улазни прикључак је наредбом:

```
pinMode(interruptPin, INPUT_PULLUP);
```

иницијално постављен на виши логички ниво повезивањем интерног pull-up отпорника према напајању. Након тога наредбом:

```
attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
```

у потпуности је дефинисан спољашњи прекид. При томе је digitalPinToInterrupt функција којом се обезбеђује да прикључак D2 преузме надзорну улогу региструјући захтев за успоставу прекида. Након тога наведено је име функције (у овом случају blink) која треба да обради прекид. На крају се наводи мод,

односно опис промене стања на прикључку D2 (у овом случају CHANGE, односно свака промена логичког нивоа) коју микроконтролер треба да „схвати“ као захтев за успоставу прекида. Платформе типа NANO прихватају још три мода: LOW (логички ниво који одговара нули), RISING (промену са нижег на виши логички ниво) и FALLING (промену са вишег на нижи логички ниво). Очигледно функција attachInterrupt захтева навођење три параметра за потпуно дефинисање спољашњег прекида. Унутар главног програма (функција loop()) постоји само једна наредба, digitalWrite(ledPin, state), којом се у зависности од вредности промељиве state укључује или искључује светлећа диода ledPin.

Прекид се активира прекидачем који је спојен између прикључка D2 и масе. При сваком притиску на њега микроконтролер преусмерава ток програма на извршење наредби унутар функције blink. Ова функција садржи само једну наредбу којом се мења вредност променљиве state = !state. Уочимо да је ова променљива дефинисана користећи кључну реч volatile. На овај начин обезбеђено је да вредност променљиве state буде смештена на неку локацију унутар RAM-а. По правилу функције за обраду прекида треба да буду што краће како би се извршавале што брже. Променљиве које се појављују унутар ових функција треба да буду дефинисане на почетку програма користећи кључну реч volatile.

10.3. Сажетак

Оптимално коришћење ресурса неког микроконтролера могуће је само уколико га ослободимо од послова као што је непрекидна провера појаве промена на појединим прикључцима. У том смислу програмска решења треба да садрже наредбе за преусмеравање активности микроконтролера као одговор на промену логичког стања

неког од прикључака условљену изван њега, односо деловањем непосредног окружења.

Употреба прекида захтева увођење посебних програмских целина, односно функцијских потпрограма који се најчешће називају сервисна рутина за обраду прекида (*Interrupt Service Routine* - ISR). У моменту настанка прекида ток програма ће се преусмерити на извршење функције од које се не очекује враћање вредности неке променљиве у главни програм. При дефинисању спољашњег прекида користи се функција attachInterrupt која захтева навођење три параметра за потпуно дефинисање спољашњег прекида, а то су ознака прилучка који прихвата прекид, име функције унутар које се исти обрађује и облик промене логичког нивоа коју програм треба да препозна као захтев за опслуживање прекида.

По правилу, функције за обраду прекида треба да буду што краће како би се извршавале што брже. Променљиве које се појављују унутар ових функција треба да буду дефинисане на почетку програма користећи кључну реч volatile. У најопштијем случају захтев за прекид може да настане као реакција на било коју промену логичког нивоа (мод CHANGE). Платформе типа NANO прихватају још три мода: LOW (логички ниво који одговара нули), RISING (промену са нижег на виши логички ниво) и FALLING (промену са вишег на нижи логички ниво).

Питања

1. Шта представља прекид у извршењу наредби унутар функције loop()?
2. Како се обезбеђује оптимално коришћење ресурса неког микроконтролера?
3. Шта се дешава у моменту обраде прекида?
4. Шта треба урадити како би се омогућило коришћење прекида типа INT0 и INT1?

11

ПОКАЗИВАЧИ

У поглављу је описана суштина показивача, односно како се дефинишу и начин њиховог коришћења. Посебан нагласак је усмерен на проблем дефинисања вредности показивача, дефинисање вредности променљиве помоћу показивача и пренос више података помоћу показивача.

Показивачи представљају посебну групу података која својом специфичном структуром и наменом у потпуности одудара од онога што смо до сада срели под појмом податак. Због тога је веома важно добро разумети шта су показивачи и како се користе да не би дошло до погрешне употребе. Њихово постојање унутар C језика представља једну од његових најважнијих карактеристика.

11.1. Дефиниција показивача

Пођимо најпре од синтаксе показивача. Њихова специфична намена условљава и специфичну синтаксу при њиховом дефинисању. За лакше разумевање дефиниције показивача наведимо један пример:

```
int *MyPointer;
```

Одмах уочавамо да се у дефиницији показивача појављује звездица. Она управо указује на чињеницу да је у питању податак типа показивач. Имајући у виду значење речи показивач, прихватимо за сада објашњење, да они не садрже вредност неке променљиве већ показују на њу. Тачније, говоре о томе где је смештена. На почетку наведеног примера појављује се кључна реч коју смо већ сретали (int) и користи се при дефиницији целобројних података. Међутим, у овом случају (int) се не односи директно на показивач већ означава тип података на које се показује. Непосредно иза звездице налази се име које, као и до сада, означава име податка, односно назив конкретног показивача.

Као што видимо, при дефиницији показивача појављују се три елемента: тип података на које се показује, звездица која означава да је у питању податак типа показивач и само име показивача. При именовању података типа показивач важе сва раније наведена правила за именовање података. Ипак, имајући у виду специфичност показивача, препоручује се да њихово име започне словима „ptr“ (од енглеске речи

pointer) како би се и на тај начин нагласило да је ово податак типа показивач. Наведимо пар примера:

```
ptrMyData
ptrName
ptrNumber
ptrADConversion
```

Уколико усвојите ову сугестију за именовање показивача свако ко чита изворни код вашег програма увак ће лако препознати овај тип података.

Навођење звездице на почетку имена података типа показивач је важно јер се тако обезбеђује да не дође до грешке при компајлирању програма. Уколико изоставите звездицу компајлер ће прихватити наведену дефиницију али ће у конкретном случају поступити као и при дефиницији сваког другог типа податка, односно, неће га третирати као показивач.

Најкритичнији део при дефинисању показивача представља спецификација његовог типа. Наведимо најпре неколико различитих типова показивача:

```
int   *ptrLicenceNumber;
char *ptrName;
long *ptrBigNumber;
float *ptrDistance;
```

У сваком од ових примера показивач је дефинисан тако да показује различити тип података. Другим речима, спецификација типа показивача је одређена типом података на које се показује. Погрешна спецификација типа показивача довела би до погрешног очитавања података. У циљу бољег разумевања овог проблема подсетимо да различити типови података могу захтевати другачији обим меморијског простора за њихово смештање као и саму структуру записа. На пример подаци типа char захтевају један бајт за смештај било којег

података овог типа за разлику од података типа int којима је за смештај података неопходно два бајта унутар одговарајуће меморије.

Намеће се питање: „Колико бајта је потребно за смештај податка типа показивач?“. Пре одговора на ово питање морамо имати на уму како изгледа меморијски простор микроконтролера. У случају ARDUINO платформи типа NANO користе се микроконтролери ATMEGA 328p. Они садрже три типа меморија. Први тип служи за смештај програма (програмска меморија). Програм мора бити сачуван у меморији микроконтролера и након престанка напајања па ово мора бити меморија типа флеш (*nonvolatile* - непроменљива). Други тип меморије користи се за смештај података који се могу мењати током извршења програма па зато припада типу SRAM (*Static Random Access Memory*). Последњи тип меморије чини EEPROM (*Electriclly Erasable Programmable Read Only Memory*). Ове меморије могу сачувати податке и након престанка напајања, али су значајно спорије у поређењу са моморијама типа флеш. Због тога се користе за смештај података који су ретко потребни или се захтевају само при иницијализацији програма и периферијских јединица.

Имајући у виду да је капацитет меморије типа SRAM, у конкретном случају, мањи од 65K при адресирању било којег дела меморијског простора биће довољан простор који није већи од 2 бајта. Подсетимо се да показивачи не садрже конкретан податак већ „показују“ на њега, односно садрже адресу меморијске локације на којој је смештен конкретан податак. Дакле, имајући у виду величину расположивог меморијског простора (65K) можемо закључити да показивачи, у конкретном случају, увек заузимају 2 бајта, без обзира на тип података на које показују. При томе је информација о величини меморијског простора који је потребан за смештај

податка на који се „показује“ дефинисана спецификацијом типа показивача. Ова вредност, у литератури се често назива скалар показивача.

Познато је да се адреса меморијске локације на којој је смештен податак често означава као lvalue (*location value*), а садржина те меморијске локације као rvalue (*register value*). Користећи ове називе при опису показивача можемо рећи да они садрже lvalue, односно показују где је смештен податак, а не шта је његова вредност, односно његова rvalue. Другим речима rvalue показивача представља lvalue податка којег показује. Уколико вас ова игра речи помало збуњује све ће вероватно постати јасније након што прочитате опис иницијализације показивача.

11.2. Иницијализација показивача

Претпоставимо да је, при иницијализацији, поинтер, односно показивач описан наредбом:

```
int *Number;
```

Нека је, након компајлирања, овом податку додељена меморијска локација са адресом 2294. Другим речима овај податак има lvalue=2294. Међутим, садржај ове меморијске локације, односно rvalue још увек није дефинисан. Може бити било која комбинација јединица и нула (XXXX) унутар меморијског простора величине два бајта са почетном адресом 2294.

адреса	садржај
2294	XX
	XX
2296	- - - -
	- -

Слика 11.1 Садржина меморијског простора након иницијализације показивача int *Number;

Уколико нам ово из било којих разлога смета могуће је, током иницијализације, унутар ових меморијских локација уписати вредност нула (NULL). Тада наведени показивач треба да се дефинише користећи наредбу:

```
int *Number = NULL;
```

Овај запис има смисла јер потпуно јасно указује да показивач још увек не показује одговарајући податак. При томе је неопходно да се на почетку програма наведе:

```
#include <stdio.h>
```

Овај запис обезбеђује да се, при компајлирању програма, позове датотека (хедер фајл) stdio.h (стандарадна улазно излазна датотека) како би се обезбедило разумевање симболичке константе NULL. Заправо, навођењем ове датотетке на почетку програма обезбеђује се да цео њен садржај буде прихваћен као део конкретног програма. Додатак .h управо означава да је ово хедер фајл (*header* - заглавље). Угласте заграде око назива хедер фајла су ту да усмере активност компајлера на процес претраге стандардног хедер фајл директоријума како би се пронашла датотека са назначеним именом.

11.3. Употреба показивача

Пре описа употребе показивача најпре претпоставимо да имамо програм који садржи наредбе:

```
int a;
int b = 5;
a = b;
```

Последња наредба, мада веома једноставна, подразумева извршење читавог низа операција које се извршавају следећим редоследом:

- најпре треба „пронаћи“ меморијску локацију чија адреса представља lvalue (адресу) променљиве b,
- затим очитати њен садржај, односно rvalue koja представља вредност променљиве b,
- након тога треба пронаћи, у табели симбола, адресу меморијске локације, односно lvalue променљиве a, отићи на ову адресу и
- копирати rvalue, односно садржај меморијске локације која је додељена за смештај променљиве b у rvalue односно меморијски простор локације која је додељена за смештај променљиве a.

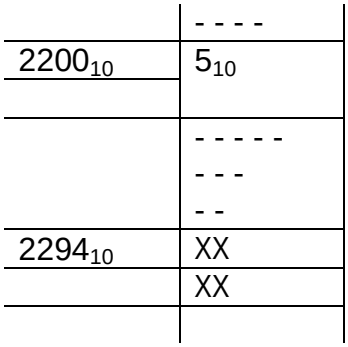
11.3.1. Дефинисање вредности показивача

Размотримо шта се дешава када се процес додељивања вредности некој променљивој остварује користећи показивач. Подсетимо се да показивач увек саджи адресу меморијске локације или NULL вредност. Намеће се питање: „Како се адреса неке меморијске локације смешта унутар меоријског простора који је додељен неком показивачу?”. За реализацију овог задатка неопходна је употреба адресног оператора (&). Појава овог оператора испред назива променљиве „упозорава“ компајлер да треба користити lvalue променљиве, а не њену rvalue. Покажимо ово на једноставном примеру. Претпоставимо да програм садржи наредбе:

```
int number = 5;
int *ptrNumber;
```

Претпоставимо, такође, да променљива number има lvalue = 2200 као и да је показивачу ptrNumber додељена lvalue = 2294. Променљива number је иницијализирана тако да њена rvalue буде 5. При томе није иницијализирана rvalue за ptrNumber. Унутар два бајта, од којих се први налази на адреси 2294, може се очитати било која вредност (XXXX). Слика 11.2

илуструје ситуацију унутар одговарајућег дела меморијског простора.



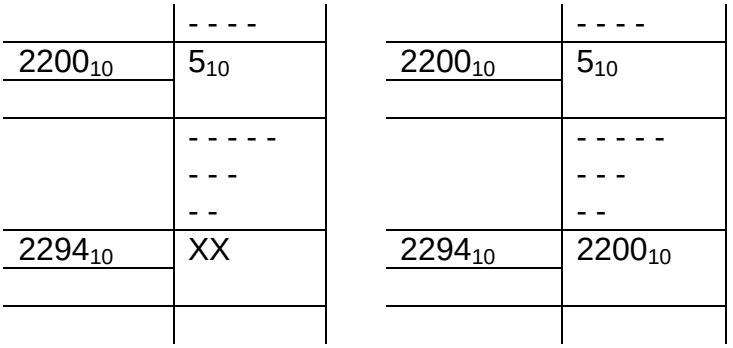
Слика 11.2 Садржина меморијског простора након иницијализације променљиве number;

Међутим, уколико се иза наведених наредби дода наредба:

```
ptrNumber = &number;          // dodeljuje se adresa
```

садржина одговарајућег меморијског простора се мења и изгледа као на слици 11.3.

Појава оператора (&) испред назива променљиве &number „упозорила“ је компајлер да при додели вредности не изврши копирање rvalue у rvalue већ да lvalue (2200), односно адресу променљиве number копира у rvalue променљиве ptrNumber. Тако сада променљива ptrNumber има lvalue = 2294, а њена rvalue је 2200, односно садржи адресу променљиве number. Другим речима, променљива ptrNumber „показује“ на променљиву number. При томе треба имати у виду да је при дефиницији показивача наведено да се односи на податке типа int па је у потпуности јасно да податак на који се показује заузима два бајта почевши од меморијске локације на коју се показује.



(а) (б)

Слика 11.3 Садржина меморијског простора пре (а) и након (б) извршења наредбе ptrNumber = &number;

11.3.2. Дефинисање вредности променљиве помоћу показивача

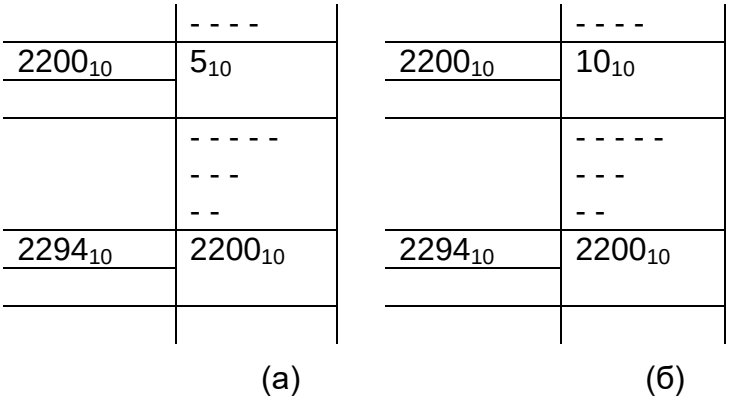
Из досадашњег текста још увек није јасно како да употребимо показивач за доделу података. Сетимо се да при дефиницији показивача испред његовог назива обавезно мора да пише звездица (*). У овом случају она представља оператор индиректног адресирања. Уколико проширимо претходну секвенцу наредби додајући *ptrNumber = 10

```
int number = 5;
int *ptrNumber;
ptrNumber = &number;
*ptrNumber = 10;
```

последња наредба се остварује кроз неколико корака:

- најпре се очитава rvalue односно вредност променљиве ptrNumber (2200),
- „одлази“ се на меморијску локацију са овом адресом и
- копира вредност 10 унутар меморијског простора величине два бајта са почетком на адреси 2200.

Уколико се питате: „Зашто унутар двобајтног меморијског простора?”, сетите се да је при дефиницији показивача наведено (int *ptrNumber;) што значи да се ради о подацима типа int који заузимају двобајтни меморијски простор. Садржина одговарајућег меморијског простора се мења и изгледа као на слици 11.4.



(а) (б)

Слика 11.4 Садржина меморијског простора пре (а) и након (б) извршења наредбе *ptrNumber = 10;

Из овог примера видимо да је садржина меморијског простора који је додељен целобројној променљивој b сада промењена индиректно користећи променљиву типа показивач.

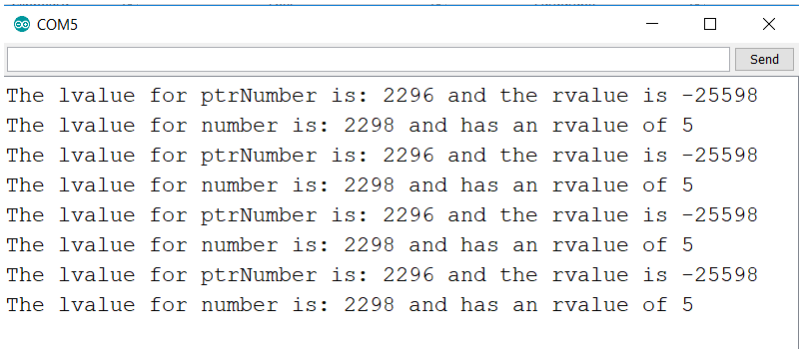
Употреба индиректног адресирања илустрована је у следећем програму (Pointer_1):

```
/* Purpose: Simple program to demonstrate using a pointer Dr. Purdum,
August 13, 2012 */
#include <stdio.h>
int counter = 0;

void setup() {
        // So we can communicate with the PC
        Serial.begin(9600);
}
```

```
void loop() {
        int number = 5;
        int *ptrNumber;
        Serial.print("The lvalue for ptrNumber is: ");
        Serial.print((long) &ptrNumber, DEC);
        Serial.print(" and the rvalue is ");
        Serial.println((long) ptrNumber, DEC);
        //      === Put new statements here!===
        Serial.print("The lvalue for number is: ");
        Serial.print((long) &number, DEC);
        Serial.print(" and has an rvalue of ");
        Serial.println((int) number, DEC);
        counter++;
        if (counter > 3) {
                Serial.flush();
                exit(0);
        }
}
```

Његовим извршењем добија се следећи испис на екрану серијског монитора:



Слика 11.5 Резултат извршења програма Pointer_1

Очигледно, програм исписује вредности lvalue и rvalue за променљиве ptrNumber и number. При томе су променљивој ptrNumber додељене вредности lvalue = 2296 и rvalue = -25598 (rvalue је настала као случајна вредност, а не као нешто што произлази из садржине програмског кода). Уколио би при иницијализацији променљиве ptrNumber писало:

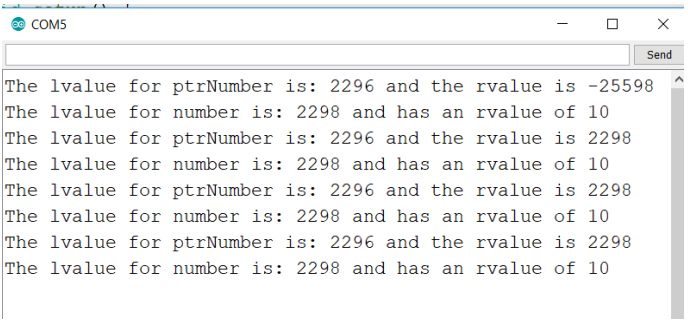
```
int *ptrNumber = NULL;
```

тада би се rvalue променљиве *ptrNumber појавила у виду броја 0, а не као нека произвољна бројна вредност.

Додајмо сада две нове наредбе иза коментара:“Put new statements here!“

```
ptrNumber = &number;
*ptrNumber = 10;
```

Измењена верзија програма даће резултат приказан на слици 11.6. Приметио да је вредност променљиве number промењена у 10. При томе rvalue променљиве ptrNumber није неки произвољни број већ означава lvalue променљиве number.



Слика 11.6 Резултат извршења програма Pointer_1 након уписа додатних наредби у програм.

Наравно, показиваче можемо користити и за доделу вредности новој променљивој. Уведимо на почетку петље loop наредбу којом се дефинише целобројна променљива k (int k;).

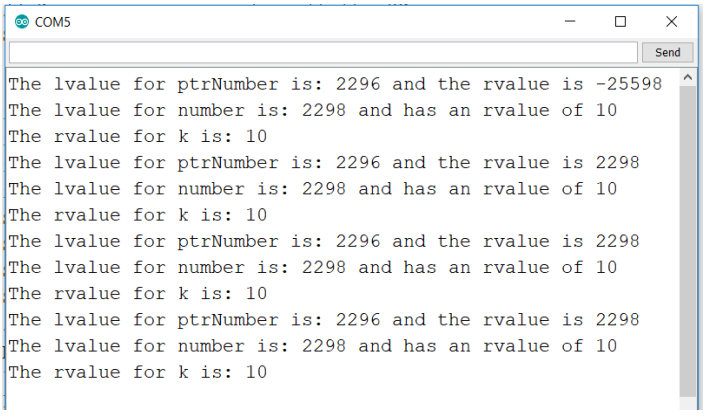
Додајмо сада у основну верзију програма Pointer_1 три нове наредбе иза коментара:“Put new statements here!“

```
ptrNumber = &number;
*ptrNumber = 10;
k = *ptrNumber;
```

Поред тога, да би уочили ефекте ове измене морају се придодати две наредбе:

```
Serial.print("The rvalue for k is: ");
Serial.println(k, DEC);
```

Измењена верзија програма даће резултат приказан на слици 11.7. Вредност променљиве number поново остаје промењена у 10, а rvalue променљиве ptrNumber није неки произвољни број већ означава lvalue, односно адресу меморијске локације на којој је смештена променљива number. Међутим, у извештају се појављује и rvalue, односно вредност променљиве k.



Слика 11.7 Резултат извршења програма Pointer_1 након додавања нових наредби.

Сумирајмо на овом месту правила за употребу показивача:

1. Показивачка променљива мора да се дефинише тако што њен назив обавезно започиње звездицом, на пример int *ptr;.
2. Показивачи не показују на податак све док им не доделимо адресу податка на који показују, односно lvalue податка на који се показује као на пример ptr = &myVariable;. У том случају lvalue променљиве myVariable постаје rvalue променљиве ptr.
3. Након иницијализације показивача омогућено је индиректно мењање rvalue вредности променљиве на коју се показује. Тако, на пример, низ наредби:

```
int myVariable;
int *ptr;
ptr = &myVariable;
*ptr = 10;
```

има ефекат доделе вредности 10 променљивој myVariable. При томе се поступак доделе заснива на примени оператора индиректног адресирања (*) и показивача ptr. Исто тако могуће је очитати вредност на коју се показује користећи индиректно адресирање:

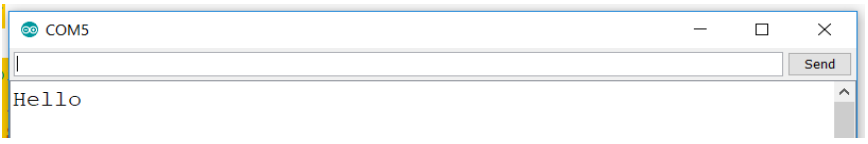
```
Serial.print(*ptr);
```

Извршењем ове наредбе вредност 10 се исписује на серијском монитору.

Усмеримо нашу пажњу, за тренутак, на податке типа array, односно низове. Поступак смештања и очитавања ових података у одговарајуће меморијске локације веома подсећа на коришћење показивача. Сваки елемент низа има позицију која је одређена целобројном вредношћу i *(i= 0,1,2,….). У наставку текста приказан је изворни код програма Pointer_2 који омогућава испис садржаја низа типа char. Његовом реализацијом, на екрану монитора, исписује се реч „Hello“ (Слика 11.8).

Изворни код програма Pointer_2:

```
/* Purpose: Display a character array using array indexes Dr. Purdum, August 14, 2012 */
void setup() {
	// So we can communicate with the PC
	Serial.begin(9600);
}
void loop() {
	char greet[6];
	int i;
	greet[0] = 'H'; // Initialize the array with some characters
	greet[1] = 'e';
	greet[2] = 'l';
	greet[3] = 'l';
	greet[4] = 'o';
	greet[5] = '\0';
	for (i = 0; i < 5; i++) {
		Serial.print(greet[i]);
	}
	Serial.flush(); // Make sure all the data is sent...
	exit(0);
}
```



Слика 11.8 Резултат извршења програма Pointer_2

Јасно је да се поступак „проналажења“ одговарајућег карактера заснива на инкрементирању индекса унутар циклуса типа FOR. Анализирајмо шта ће се десити ако наредбу унутар FOR циклуса заменимо наредбом:

```
Serial.print(*(greet+i));.
```

Реализацијом програма, након ове измене, на екрану монитора би се добио исти резултат, односно испис речи „Hello“. Покушајмо сада да уместо ове наредбе напишемо:

```
Serial.print(*(greet+i * sizeof(char)));.
```

И у овом случају, испис на екрану монитора остаје исти. Ово се објашњава чињеницом да самостална употреба назива низа има исти ефекат као и употреба вредности lvalue која одговара том низу.

Погледајмо за тренутак како су елементи низа смештени у меморијском простору. Претпоставимо да је, као на слици 11.9, овај низ смештен почевши од меморијске локације са адресом 2200 (ово значи да је његова вредност lvalue = 2200).

	- - - -
2200_{10}	H
2201_{10}	e
2202_{10}	l
2203_{10}	l
2204_{10}	o
2205_{10}	‘\0’
	- - - -

Слика 11.9 Садржина меморијског простора у којем је смештен садржај низа greet; типа char.

Анализирајмо шта ће се десити реализацијом наредбе:

```
Serial.print(*(greet+i));.
```

Након првог пролаза кроз циклус типа FOR индекс i има вредност 0 па се наредба извршава у корацима који се могу описати на следећи начин:

```
Serial.print(*(greet + 0));.
Serial.print(*(2200 + 0));.
Serial.print(*(2200));.
```

Опертаор индиректног адресирања, (*), условљава одлазак на меморијску локацију са адресом 2200 где се врши „дохватање“ (*fetch*) карактера ‘H’.

Након следећег пролаза кроз циклус типа FOR индекс i има вредност 1 па се наредба извршава у корацима који се могу описати на следећи начин:

```
Serial.print(*(greet + 1));.
Serial.print(*(2200 + 1));.
Serial.print(*(2201));.
```

Адреса меморијске локације је увећана за један (променљива типа char заузима само један бајт) па се врши „дохватање“ карактера ‘e’. Поступак се понавља до краја FOR циклуса.

Друга варијација наредбе Serial.print(greet[i]);, односно

```
Serial.print(*(greet+i * sizeof(char)));
```

имаће исти ефекат као и претходбна. Зашто? Оператор sizeof() враћа број бајта који су неопходни за смештај података чији је тип наведен у загради. Због тога се ова наредба, у конкретном случају, може описати следећим корацима:

```
Serial.print(*(greet + i * 1));
Serial.print(*(greet + 0 * 1));
Serial.print(*(2200 + 0 ));
Serial.print(*(2200));
```

тако да се као резултат исписује слово ‘H’. Након другог пролаза кроз циклус наредба се извршава у следећим корацима:

```
Serial.print(*(greet + i * 1));
Serial.print(*(greet + 1 * 1));
Serial.print(*(2200 + 1));
Serial.print(*(2201));
```

тако да се као резултат исписује слово ‘e’. Поступак се понавља на истоветан начин све до испуњења услова за напуштање FOR циклуса.

Опис реализације приказаних варијација наредбе за испис низа потврђује да самостална употреба назива низа greet (нагласимо, назив без угластих заграда) има исти ефекат као и употреба вредности lvalue низа greet[].

Уведимо сада, у основој верзији програма pointer_2, измену којом ћемо дефинисати показивач:

```
void loop() {
        char greet[6];
        char *ptr;
int i;
        greet[0] = 'H';
        greet[1] = 'e';
        greet[2] = 'l';
        greet[3] = 'l';
        greet[4] = 'o';
        greet[5] = '\0';
        ptr = greet; // Initialize the pointer
        for (i = 0; i < 5; i++) {
                Serial.print(*ptr++);
        }
        Serial.flush(); // Make sure all the data is sent...
        exit(0);()
}
```

Извршењем програма поново се добија исти испис на екарну. Наредбом

```
ptr = greet;
```

вредност lvalue низа greet постаје rvalue показивача ptr. С обзиром да, у овом случају, податак на који се показује представља низ, компајлер очитавањем имена низа

аутоматски преузима његову lvalue вредност, па нема потребе да се испред имена наводи адресни оператор (&). Наредба:

```
Serial.print(*ptr++);
```

користи индиректан оператор (*) да дохвати елементе низа greet. С обзиром да је при првом пролазу кроз FOR циклус ptr = 2200 на екрану монитора се исписује слово ‘H’. Инкрементирање показивача ptr се остварује применом постфиксног оператора због чега је у наредном проласку кроз FOR циклус ptr = 2201 па се на екрану монитора исписује слово ‘e’. Видимо да све три варијанте програма дају исти резултат.

Нагласимо да оператор ++ увећава вредност показивача за 1, 2 или 4 у складу са типом податка на који показује (скалар показивача).

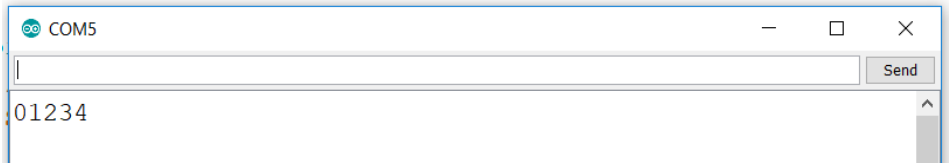
Направимо неколико минорних измена у програму Pointer_2 тако да омогућимо испис низа којег чине цели бројеви. Садржај ове вариајнте дат је у виду изворног кода са називом Pointer_3.

Изворни код програма Pointer_3:

```
/*Purpose: Display an int array using array indexes Dr. Purdum, Aug. 13, 2012*/
    void setup() {
        Serial.begin(9600);
    }
    void loop() {
        int greet[6];       // Notice this is an int now
        int *ptr;  // ...as is this
        int i;
        greet[0] = 0;       // Numbers now...
        greet[1] = 1;
        greet[2] = 2;
        greet[3] = 3;
        greet[4] = 4;
        greet[5] = 5;
```

```
        ptr = greet;
        for (i = 0; i < 5; i++) {
                Serial.print(greet[i]);
        }
        Serial.flush(); // Make sure all the data is sent...
        exit(0);
}
```

Резутат извршења овог програма се види на слици 11.10. За разлику од предходне варијанте програма, инкрементирањем показивача ptr, адреса податка који се дохвата, сваки пут се увећава за два тако да је након првог проласка FOR циклуса ptr = 2202, а не 2001 као у случају податка типа char.



Слика 11.10 Резултат извршења програма Pointer_3

11.3.3. Пренос више података помоћу показивача

Извршење функцијских програма, у случају језика C писаног за ARDUINO платформе, омогућава враћање само једног резултата. У случају када је потребно да се након извршења функцијског програма врати већи број вредности морамо користити показиваче. Следећи програм (D2S_konv) управо илуструје једну такву ситуацију. У конкретном случају програм остварује конверзију правоуглих координата неке тачке (*x, y, z*), у тродимензионалном простору, у сферне координате (*α, β, ρ*). При томе су, (*x, y*) координате у хоризонталној равни, (*z*) висина, (*α*) - азимут, (*β*) – *елевација, а* (*ρ*) – коса даљина. Њихови међусобни односи су дефинисани једначинама:

$$\alpha = \sin^{-1}(y/x)$$
$$\beta = \sin^{-1}(y/\rho)$$
$$\rho = \sqrt{x^2 + y^2 + z^2}\ .$$

Изворни код програма D2S_konv:

```
/*Namena: Racuna sferne na osnovu pravoglih kordinata dtaPrv[x, y, z]*/
#include <stdio.h>
#include <math.h>
#define rd2dg 180/PI
float dtaPrv[]={-600.,30.,3.};
void setup() {
        Serial.begin(9600);
}
void loop() {
        float alfa;     // ugao u horizontalnoj ravni
        float beta;     // ugao u vertikalnoj ravni
        float distance; // kosa daljina
        int retVal;
        // Koordinate u pravoglom koordinatnom sistemu
        Serial.println("Pozicija u metrima ");
        Serial.println("     (X,Y,Z):");
        Serial.println(dtaPrv[0]);
        Serial.println(dtaPrv[1]);
        Serial.println(dtaPrv[2]);
        //  Sadrzaj lokacija “Koordinate u sfernom koordinatnom sistemu”
        // pre konverzije
        Serial.println("");
        Serial.println("Pre ivrsenja programa Sferne koordinate su:");
        Serial.print("Ugao alfa = ");
        Serial.print(alfa);
        Serial.println(", u stepenima");
        Serial.print("Ugao beta = ");
        Serial.print(beta);
```

```
            Serial.println(", u stepenima");
            Serial.print("Udaljenost = ");
            Serial.print(distance);
            Serial.println(", u metrima");
            // --------- Poziva podprogram za konverziju koordinata
            retVal= CalculateSferCord(dtaPrv, &alfa, &beta, &distance);
            // --------- Koordinate u sfernom koordinatnom sistemu
            Serial.println("");
            Serial.println("Sferne koordinate su:");
            Serial.print("Ugao alfa = ");
            Serial.print(alfa);
            Serial.println(", u stepenima");
            Serial.print("Ugao beta = ");
            Serial.print(beta);
            Serial.println(", u stepenima");
            Serial.print("Udaljenost = ");
            Serial.print(distance);
            Serial.println(", u metrima");
            Serial.flush();
            exit(0);
}
/*
*  Namena: izracunava sferne koordinate
*  Parametri:     float mer[] - niz vrednosti dtaPrv[x, y, z]
*                 float *alfD – pokazivac na azimut
*                 float *btaD – pokazivac na elevaciju
*                 float *roD – pokazivac na kosu daljinu
*  Vraca: celeobrojnu vrednost 0 */
int CalculateSferCord(float mer[], float *alfD, float *btaD, float *roD){
      *alfD= rd2dg*atan(mer[1]/mer[0]);                              // azimut
      *roD = sqrt(pow(mer[0],2)+pow(mer[1],2)+pow(mer[2],2)); // daljina
      *btaD= rd2dg*asin(mer[2]/(*roD));                          // elevacija
      return 0;
}
```

Анализом изворног кода видимо да су правоугле кординате уведене у програм као низ float dtaPrv[]. При томе су, правоугле координате регистроване у метрима. На самом почетку програма, поред библиотеке stdio.h наведена је и математичка библиотека math.h. Уз то је дефинисана и константа rd2dg којом се врши конверзија угловних резултата из радијана у степене. На почетку void loop функције дефинисане су четири променљиве:

```
float alfa;       // ugao u horizontalnoj ravni
float beta;       // ugao u vertikalnoj ravni
float distance;   // kosa daljina
int retVal;       // celobrojna vrednost koja se vraca nakon izvrsenja programa
```

како би се омогућила додела вредности: азимут (alfa), елевација (beta), коса даљина (distance) и повратна вредност (retVal), односно резултат који се враћа у главни програм директно након процеса обраде података у функцијском подпрограму D2S_konv. Овај функцијски подпрограм садржи четири параметра (float mer[], float *alfD, float *btaD, float *roD). Први означава низ правоуглих координата (*x, y, z*), а други, трећи и четврти представљају показиваче на азимут, елевацију и косу даљину. Уочимо како је овај функцијски програм позван унутар loop() функције:

```
retVal= CalculateSferCord(dtaPrv, &alfa, &beta, &distance);.
```

Прво је наведено име низа у којем су смештене вредности правоуглих координата, а затим адресе меморијских локација на којима треба да се сместе подаци о азимуту, елевацији и косој даљини. Важно је истаћи да се навођењем имена низа у функцијски подпрограм преноси његова адреса (lvalue), а не вредности елемената које садржи.

Ако сада погледамо како су дефинисани параметри функцијског програма:

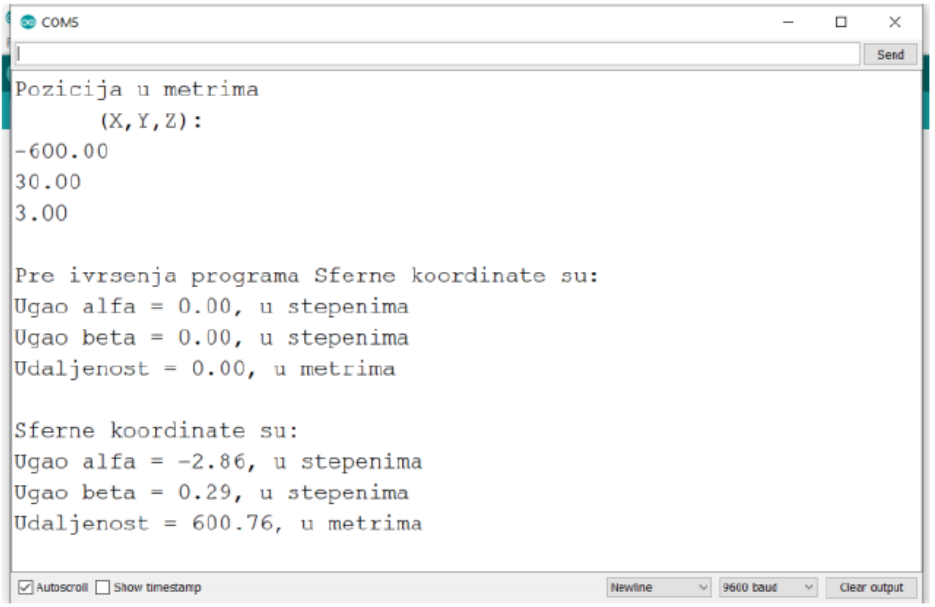
```
int CalculateSferCord(float mer[], float *alfD, float *btaD, float *roD)
```

уочавамо да су на позицијама параметара азимут, елевација и коса даљина, за разлику од позива функције у главном програму, сада наведени подаци типа показивач (float *alfD, float *btaD, float *roD). Наравно, њихови називи недвосмислено указују да се односе на азимут, елевацију и косу даљину.

На овај начин су адресе меморијских локација (&alfa, &beta, &distance), у које треба да се упишу вредности азимута, елевације и косе даљине, „саопштене“ одговарајућим показивачима (*alfD, *btaD, *roD). Ово је заправо исто као да смо написали:

int CalculateSferCord (float mer[],*alfD= &alfa, *btaD= &beta, *roD=&distance).

Очигледно, захваљујући примени показивача, вредности сферних координата су смештене на одговарајуће локације, током процеса извршења подпрограма. Резултат извршења програма је приказан на слици 11.11. Уочимо да пре позива функцијског програма D2S_konv меморијске локације са адресама (&alfa, &beta, &distance) садрже нулу записану у децималном формату, а након тога израчунате вредности сферних координата.



Слика 11.11 Резултат извршења програма D2S_konv.

Уколико пажљиво анализирамо резултате израчунавања може се уочити да је заправо добијен погрешан резултат за азимут. У чему је проблем? У питању је природа функције atan (*arkus tanges*). Познато је да она има основну област дефинисаности од -90^o до +90^o. Уколико желимо да се добију исправни резултати и за преостале углове морамо заменити линију (у програмској функцији D2S_konv)

```
*alfD= rd2dg*atan(mer[1]/mer[0]);
```

тако да пише

```
*alfD= rd2dg*atan2(mer[1],mer[0]);.
```

Математичка библиотека math.h садржи обе функције. Имајући у виду потребу да се конверзија остварује у комплетном простору боље је тада користити функцију atan2. У конкретноим случају након ове измене добио би се коректан резултат alfa= 180-2,86= 177,14.

11.4. Сажетак

При дефиницији показивача појављују се три елемента: тип података на које се показује, звездица која означава да је у питању податак типа показивач и само име показивача. Препоручује се да њихово име започне словима „ptr“ (од енглеске речи *pointer*) како би се и на тај начин нагласило да је ово податак типа показивач.

Показивачи не садрже вредност неке променљиве већ показују на њу. Тачније, говоре о томе где је смештена. Спецификација типа показивача је одређена типом података на које се показује. Различити типови података могу захтевати другачији обим меморијског простора за њихово смештање као и саму структуру записа.

Имајући у виду величину расположивог меморијског простора (65K) показивачи, у конкретном случају, увек заузимају 2 бајта, без обзира на тип података на које показују. Величина меморијског простора који је потребан за смештај податка на који се „показује“ дефинисана је спецификацијом типа показивача. Ова вредност, у литератури се често назива скалар показивача.

При опису показивача можемо рећи да они садрже lvalue, односно показују где је смештен податак, а не шта је његова вредност, односно његова rvalue. Другим речима rvalue показивача представља lvalue податка којег показује.

У случају када је потребно да се након извршења функцијског програма врати већи број вредности морамо користити показиваче. При томе се при позиву функције, у главном програму, као параметри наводе име низа у којем су смештене вредности улазних података, а затим адресе меморијских локација на којима треба да се сместе подаци који се враћају након процеса обраде у функцијском програму.

За разлику од позива функције у главном програму, при дефинисању саме функције као параметри се наводе име низа и називи показивача на адресе података који се враћају у главни програм.

На овај начин су адресе меморијских локација у које треба да се упишу вредности повратних података „саопштене“ одговарајућим показивачима.

Питања:

1. Како се, у програму, дефинише показивач?
2. Колика је величина меморијског простора неопходна за смештај података типа показивач?
3. Шта је скалар показивача и од чега зависи његова вредност?
4. Како се адреса неке меморијске локације смешта унутар меморијског простора који је додељен неком показивачу?
5. Објасните поступак доделе података путем индиректног адресирања.
6. Објасните поступак преноса података у главни програм у случају када функцијски подпрограм генерише више резултата.

12

ОПТИМИЗАЦИЈА ARDUINO ПРОГРАМА

У овом поглављу, централно питање, представља проблем оптимизације како хардверских тако и софтверских ресурса, пре свега, из аспекта брзине обраде података, односно у складу са захтевима њихове обраде у реалном времену.

Примена микроконтролера у пракси подразумева оптимално коришћење свих његових ресурса како у погледу могућности коришћења расположивих периферија (хардверски ресурси) тако и рад у реалном времену. Ово практично значи да програмска решења треба да минимизирају време приступа расположивим периферијским јединицама и рачунско време неопходно за генерисање резултата обраде података у циљу генерисања управљања (активирање релеја и слично).

12.1. Оптимално коришћење хардверских ресурса

Видели смо да је у случајевима када се захтевало да микроконтролер омогући укључење или искључење светлеће диоде у неком временском интервалу коришћена наредба

```
delay(ms);
```

која обезбеђује да конкретно стање траје у складу са вредношћу променљиве ms. При томе, контролер „стоји у месту“, односно чека се да наредна инструкција буде извршена тек након истека овог времена. Проблем постаје израженији уколико је ово време дуже, а истовремено треба да се користе услуге неке од периферних јединица или надзире промена стања на неком од расположивих улаза.

Уколико је прихватљиво извесно толеранцијско одступање, односно процес може да траје и незнатно дуже у односу на време дефинисано променљивом ms боље је користити програмско решење којим се стално опсервира колико је времена протекло од почетка извршења конкретног програма. Програмска подршка за ARDUINO платформе садржи две функције које нуде ову могућност millis() и micros(). Прва омогућава мерење времена у милисекундама, а друга у микросекундама.

Уколико се унутар конкретног програмског решења уведу наредбе:

```
long startTime = micros();
{
        // blok naredbi
}
long endTime = micros();
long microseconds = endTime-startTime;
```

променљива microseconds својом вредношћу указује на време које је неопходно за извршење конкретног блока наредби. Очигледно, могуће је увести мерење времена у односу на тренутак када је започео неки програмски процес или активност у окружењу микроконтролера.

Претпоставимо да је, у случају програма којим се регулише режим рада светлеће диоде, пре уласка у void loop() секвенцу одређена вредност startTime променљиве и да је диода искључена. Уколико желимо да диода емитује светлост након времена TimeOff неопходно је увести следеће наредбе унутар void loop() секвенце

```
If ((micros()-startTime) >= TimeOff)
{       // naredba za ukljucenje diode
        startTime = micros()    // za merenje novog intervala
}
// prva sledeca naredba izvan bloka
```

Аналогним поступком треба да се дефинише блок наредби за поновно искључење диоде. Очигледно, програм „не стоји у месту“ већ се извршава сукцесивно унутар void loop() секвенце све док не буде испуњен услов (micros()-startTime) >= TimeOff.

Обратите пажњу да у случају када се жели наизменична промена режима рада светлеће диоде (светли / не светли)

треба унети извесне промене како не би настао проблем. Потребно је увести додатну променљиву LedFlag којом се одређује тренутно стање светлеће диоде (светли / не светли). Овим се избегава конфузија у погледу тестирања услова. На пример, ако је диода тренутно искључена онда нема потребе да се испитује услов (micros()-startTime) >= TimeOn. Одговарајући програм, поред осталих, треба да садржи следећи низ наредби

```
.....
.....
If ((LedFlag = 0) & (micros()-startTime) >= TimeOff));
{       // naredba za ukljucenje diode
        LedFlag = 1;
        startTime = micros()    // za merenje novog intervala
}
else
If ((micros()-startTime) >= TimeOn);
{       // naredba za isukljucenje diode
        LedFlag = 0;
        startTime = micros()    // za merenje novog intervala
}
// prva sledeca naredba izvan If bloka
```

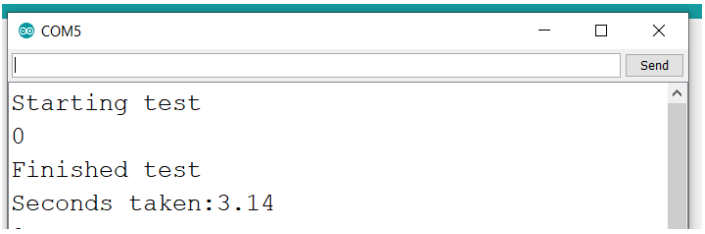
12.2. Оптимизација рачунксих операција

Управљање у реалном времену захтева да се рачунања изведу у што краћем времену како не би дошло до кашњења, а самим тим и погоршања карактеристика конкретног система. У том смислу, кад год је могуће, треба користити целобројну аритметику јер се математичке операције тада извршавају брже. Покажимо то на једноставном примеру програма IntegerTest:

```
/* Program:     IntegerTest */
```

```
void setup() {
        Serial.begin(9600);
        Serial.println("Starting test");
        long startTime = millis();
        // test sequence
        long i = 0;
        long j = 0;
        for (i=0; i<2000000; i++)
        {
                j= i + i*10;
                if (j>10) j=0;
        }
        // end of test code
        long endTime = millis();
        float seconds = float(endTime-startTime) /1000.0;
        Serial.println(j);
        Serial.println("Finished test");
        Serial.print("Seconds taken:");
        Serial.println(seconds);
}
void loop() { }
```

Програм садржи једноставан блок наредби које се извршавају два милиона пута унутар for циклуса. Његовим извршењем добија се резултат приказан на слици 12.1. Видимо да се све операције у вези са контролом циклуса и рачунских радњи обаве за 3,14 секунди.



Слика 12.1 Резултат извршења програма IntegerTest

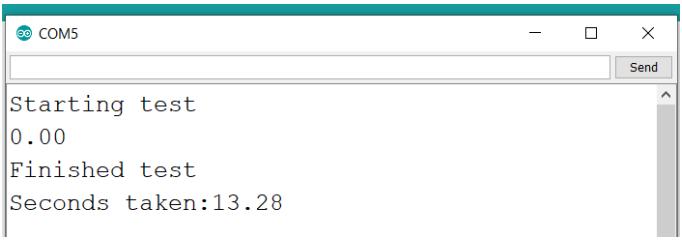
Уколико, у истом програму, променимо тип променљиве тако што наредбу (осенчена програмска линија)

```
long j = 0;
```

заменимо наредбом

```
float j = 0.0;    // float deklaracija umesto long
```

извршењем програма добија се резултат приказан на слици 12.2. Добијени резултат јасно указује да извршење програма траје знатно дуже, односно 13,28 секунди.



Слика 12.2 Резултат извршења програма IntegerTest након промене типа променљиве.

Примена *look-up* табела представља следећу могућност да се убрза рад програма. Претпоставимо да је потребно изгенерисати сигнал који се мења у духу синусне функције (y=sin(x)). Уколико су осам бита довољни да се постигне задовољавајућа резолуција онда одговарајућа променљива треба да буде типа byte јер се на тај начин заузима мање меморијског простора и убрзавају сва рачунања (осмобитни процесор). Претпоставимо да овај сигнал треба да се генерише на принципу PWM генератора (однос сигнал пауза се мења задавањем одговарајуће променљиве у опсегу од 0 до 255). Онда све вредности сигнала треба да буду веће од нуле, односно треба их рачунати по формули:

y= (1+sin(x))/2.

Надаље, претпоставимо да је довољно да угловна резолуција генерисаног сигнала буде dalf=PI/32. Имајући у виду

периодичност синусног сигнала тада је сасвим довољно да се формира низ од 33 елемента користећи следећу програмску секвенцу:

```
byte val[33]; // niz od 33 elementa
int i=0;
float dalf=PI/32;
void setup() {
        Serial.begin(9600);
        for(float alf=(-PI/2); alf<(PI/2+dalf); alf=alf+dalf){
                val[i]=byte(round(127.5*(sin(alf)+1)));
                i++;
        }
        i=0;
}
```

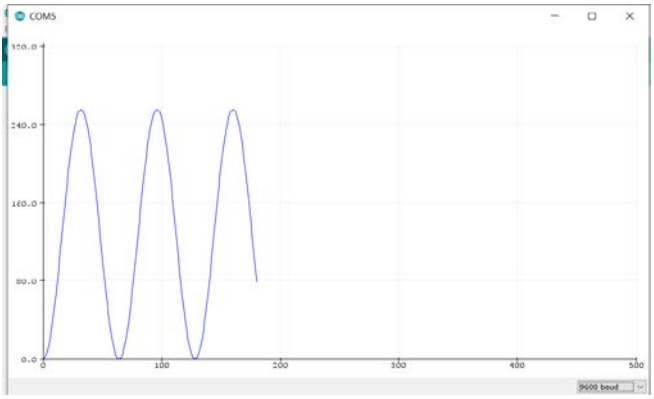
Из приложене секвенце се види да су најпре дефинисани низ (byte val[33]), целобројна променљива (int i=0;) и угловна резолуција сигнала (float dalf=PI/32;). Након тога, унутар void setup() функције, користећи циклус типа FOR, израчунавају се сви елементи низа val[33]. Цео програм садржи следеће линије кода:

```
/* Niz celobrojnih vrednosti sa medjusobnim odnosima koji odgovaraju funkciji y=sin(x) */
byte val[33]; // niz od 33 elementa
int i=0;
float dalf=PI/32;
void setup() {
        Serial.begin(9600);
        for(float alf=(-PI/2); alf<(PI/2+dalf); alf=alf+dalf){
                val[i]=byte(round(127.5*(sin(alf)+1)));
                i++;
        }
        i=0;
}
```

```
void loop() {
        while(i<32){
                Serial.println(val[i]);
                delay(100);
                i++;
        }
        delay(100);
        while(i>0){
                Serial.println(val[i]);
                delay(100);
                i--;
        }
}
```

Треба уочити да се израчунавају вредности за углове у интервалу од (-PI/2) до (PI/2) што је сасвим довољно јер се исте вредности понављају у обрнутом следу на интервалу (PI/2) до (PI). На крају void setup() секвенце поново се дефинише вредност целобројне променљиве i=0; како би се припремио улазак у циклус типа while на почетку void loop() секвенце. Унутар овог циклуса приказује се првих 32 вредности низа у интервалима од 100 милисекунди (елементи на позицијама од 0 до 31). Након изласка из првог while циклуса следи још једно кашњење од 100 милисекунди како би се извршила припрема за приказ последњег (32.) елемента низа, одмах по уласку у наредни while циклус. Овај пут исчитавање елемената низа остварује се обрнутим редом, од позиције 32 до позиције 1. Након изласка из овог циклуса целобројна променљива има вредност i=0. Самим тим створени су услови да се процес понови одласком на почетак void loop() функције.

Резултат извршења програма, у виду графичког приказа, добијен је коришћењем опције Serial ploter и приказан је на слици 12.3. Јасно се уочава синусна природа промене добијених вредности у интервалу од 0 до 255.



Слика 12.3 Резултат извршења програма за генерисање синусног сигнала.

Очигледно, суштина оптимизације се огледа у чињеници да се све вредности рачунају само једном унутар void setup() функције. Након тога само се исчитавају при сваком проласку кроз void loop() функцију. Променом вредности унутар функције delay добија се низ вредности које се мењају брже или спорије у складу са конкретном потребом.

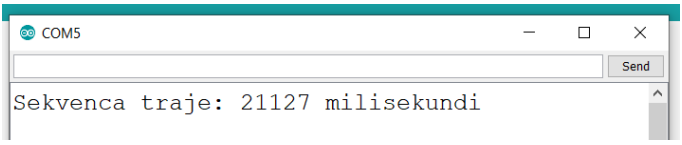
12.3. Оптимизација READ/WRITE процеса

Већ је познато да се промена стања на пиновима остварује применом наредбе digitalWrite. Овај приступ омогућава брзо и лако писање програма али има и своје лоше стране. Уколико је потребно променити стање на већем броју пинова потребно је то учинити посебно за сваки од њих. Извршење функцијског програма који се „крије“ иза ове наредбе сваки пут троши неко време па је разумљиво да се повећањем захтева у погледу промена одговарајућих стања троши све више времена, односно успорава пролазак кроз void loop() секвенцу. Са намером да се стекне утисак о величини времена које се троши у овом процесу креиран је једноставан програм DigitalOutClasic у којем се промена стања на пину 10

остварује два милиона пута унутар одговарајућег while циклуса.

```
/*
* Program:      DigitalOutClasic
* Promena stanja na pinu 10 pomocu naredbe digitalWrite
*/
long i= 1;
void setup() {
        Serial.begin(9600);
        long startTime= micros();
        while (true){
                digitalWrite(10,HIGH);
                digitalWrite(10,LOW);
                i++;
                if (i>2000000)
                        break;
        }
        long endTime= micros();
        Serial.print("Sekvenca traje: ");
        Serial.print((endTime-startTime)/1000);
        Serial.print(" milisekundi");
}
void loop() { }
```

Његовим извршењем добија се резултат који је приказан на слици 12.4.



12.4 Резултат извршења programa DigitalOutCLASIC

Цео поступак може да се обави значајно брже уколико се приступ конкретном пину оствари преко унутрашњих регистара микроконтролера.

Да би ово објаснили рецимо најпре нешто о груписању пинова (прикључака), односно организацији улазно излазних портова (група пинова) као и припадајућим регистрима који дефинишу начин њиховог коришћења.

Начин груписања пинова у портове приказан је на слици 12.5. Уочимо да су пинови намењени за пренос дигиталних информација груписани у два порта D (пинови D0-D7) и B (пинови D8-D13). Пинови за пренос аналогних информација су груписани унутар порта C (пинови A0- A5).

Port D							
D7	D6	D5	D4	D3	D2	D1	D0

Port B							
		D13	D12	D11	D10	D9	D8

Port C							
		A5	A4	A3	A2	A1	A0

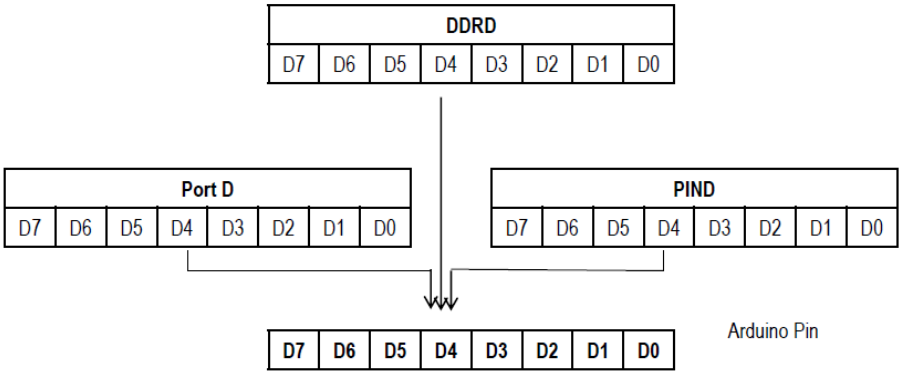
Слика 12.5 Начин груписања пинова унутар портова улазноизлазних портова.

Сваки од портова се контролише преко три регистра. На слици 12.6 су приказани регистри за контролу порта D.

Први од њих *data direction register D* (DDRD) садржи осам бита од којих сваки својим садржајем одређује смер преноса информације за одговарајући пин унутар порта D. Тако, на пример, уколио желимо да пин D0 буде коришћен као излаз, бит на позицији нула унутар DDRD регистра мора да садржи логичку вредност 1. У супротном случају (логичка 0) овај пин ће бити дефинисан као улаз. Већ позната, ARDUINO функција pinMode управо користи овај регистар како би се дефинисала употреба неког пина (као улаз или као излаз).

Регистар са ознаком PORTD користи се за дефинисање логичког нивоа (1 или 0) који треба да се појави на одговарајућем пину.

Последњи регистар *port input D* (PIND) служи за очитавање логичких нивоа на улазним пиновима.



Слика 12.6 Регистри који се користе у раду са портом D.

Сваки од три порта има своја три регистра. Тако, у случају порта B они имају ознаке DDRB, PORTB и PINB, а порту C одговарају регистри са ознакама DDRC, PORTC и PINC.

Са намером да се провери колико је краће извршење програма уколико се приступ пину оствари преко унутрашњих регистара микроконтролера тестиран је резултат иизвршења програма DigitalOutFast:

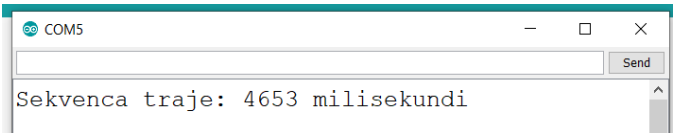
```
/* Program DigitalOutFast */
long i= 1;
void setup() {
        // put your setup code here, to run once:
        Serial.begin(9600);
        DDRB= B00000100;
        long startTime= micros();
        while (i<2000000){
                PORTB= B00000100;
```

```
            PORTB= B00000000;
            i++;
        }
        long endTime= micros();
        Serial.print("Sekvenca traje: ");
        Serial.print((endTime-startTime)/1000);
        Serial.print(" milisekundi");
}
void loop() { }
```

Уочимо да наредба DDRB= B00000100; дефинише пин D2 као излаз, односно за слање податка изван микроконтролера. Узастопна промена логичког стања на овом пину остварује се наредбама

```
PORTB= B00000100;
PORTB= B00000000;
```

2.000.000 пута унутар while циклуса. Извршењем овог програма на монитору се добија порука приказана на слици 12.7. Очигледно, наизменична промена логичког нивоа на конкретном пину извршва се знатмо брже (4.653 милисекунде) него у претходном случају када су коришћене наредбе засноване на функцији digitalWrite (21.127 милисекунди)

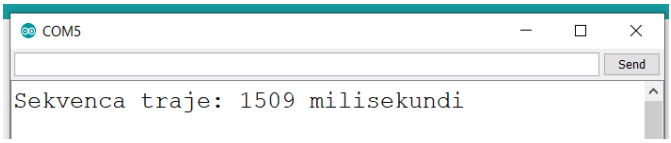


Слика 12.7 Резултат извршења програма DigitalOutFast.

За стицање увида у то колико заправо траје извршење наредби за промену лигичког стања на пину D2 исти програм је извршен тако што су одговарајуће наредбе претворене у коментар:

```
// PORTB= B00000100;          stanje se ne menja
// PORTB= B00000000;          stanje se ne menja
```

па се самим тим и не извршавају. Добијени резултат је приказан на слици 12.8. Као што се види извршење осталих наредби захтева 1.509 милисекунди у односу на укупних 4.653 милисекунде. Значи промена логичког стања на пину D2 на овај начин се оствари 2.000.000 пута за приближно три секунде што је значајно брже у односу на програмску реализацију DigitalOutClasic (21.127 милисекунди).



Слика 12.8 Резултат извршења програма DigitalOutFast (без промена логичког стања на пину D2)

Исти метод се може користити и за брже очитавање логичких стања на улазним пиновима. При томе је за очитавање врло кратких импулса боље користити решења заснована на обради прекида.

Претпоставимо да је потребно симултано очитавање стања на више пинова унутар истог порта. Тада директан приступ омогућава значајно брже извршење програма у односу на примену већ познате фунције digitalRead.

Програм DigitalInputFast управо омогућава симултано очитавање стања на свих осам пионова (D8 - D13) унутар порта B.

```
/* Program: DigitalInputFast */
byte state = 0;
void setup(){
        DDRB = B00000000; // all inputs
        Serial.begin(9600);
}
```

```
void loop(){
        Serial.println(PINB, 2);
        delay(1000);
}
```

Регистар DDRB поставља све битове у стање логичке нуле означавајући на тај начин да су сви пинови порта B дефинисани као улаз, односно омогућавају унос података. Унутар void loop() секвенце очитани подаци се приказују у виду бинарног податка применом наредбе

```
Serial.println(PINB, 2);
```

Број 2 управо сигнализира да се захтева испис у бинарном формату. Треба имати у виду да је извршење функције Serial.println веома споро (неколико десетака милисекунди) у поређењу са операцијом очитавања стања па је треба избећи уколико је време критичан ресурс, односно уколико се захтева брзо извршење програма.

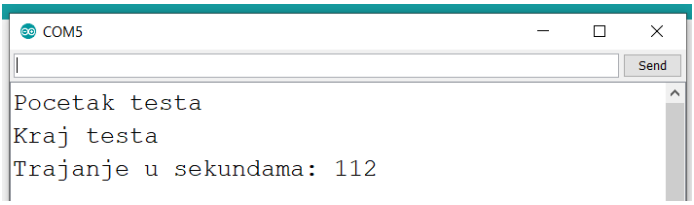
12.4. Убрзано очитавање аналогних улаза

Најпре анализирајмо колико траје поступак аналогнодигиталне конверзије када се користи већ добро позната функција analogRead. Ова анализа се заснива на мерењу времена извршења 1.000.000 узастопних AD конверзија (програм AnalogReadClasic). Конкретан програм садржи следеће наредбе:

```
/*  Program: AnalogReadClasic - Testira trajanje analgone konverzije */
void setup() {
        Serial.begin(9600);
        while(!Serial){};
        Serial.println("Pocetak testa");
        long startTime = millis();
        // test sekvenca
        long i = 0;
```

```
        for (i=0; i<1000000; i++)
                {analogRead(A0);
        } // kraj test sekvence
        long endTime= millis();
        Serial.println("Kraj testa");
        Serial.print("Trajanje u sekundama: ");
        Serial.println((endTime-startTime)/1000);
}
void loop() { }
```

Његовим извршењем добија се резултат, приказан на слици 12.9, који указује да је за 1.000.000 узастопних AD конверзија неопходно 112 секунди.



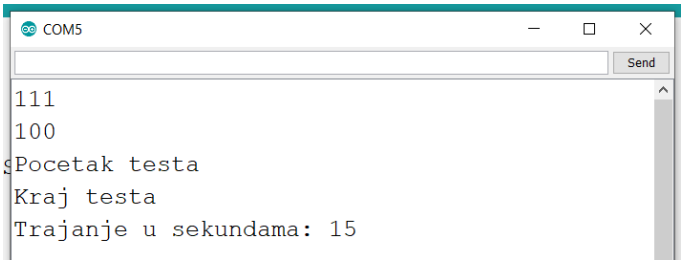
12.9 Резултат извршења програма AnalogReadClasic

Принцип AD конверзије која се користи у случају конкретног микроконтролера (сукцесивна апроксимација) већ је описан у седмом поглављу. Не улазећи превише у детаље архитектуре микроконтролера додајмо само да се процес униформног одабирања сигнала одвија под контролом интерног тајмера и припадајућег прескалера. Уколико се постојећи програм модификује тако што се промене режими рада ових делова интерне архитектуре микроконтролера може се остварити знатно краћи процес AD конверзије. Конкретне измене приказане су осенченим линијама у програму AnalogReadFast:

```
/*  Program: AnalogReadFast - Testira trajanje analgone konverzije modifikovana verzija AnalogReadClasic */
        const byte PS_128 = (1<< ADPS2) | (1<< ADPS1) |(1<< ADPS0);
```

```
const byte PS_16 = (1<< ADPS2);
void setup() {
        ADCSRA &= ~PS_128;  // remove prescale of 128
        ADCSRA |= PS_16;    // add prescale of 16 (1MHz)
        Serial.begin(9600);
        while(!Serial){};
        Serial.println(PS_128,2);
        Serial.println(PS_16,2);
        Serial.println("Pocetak testa");
        long startTime = millis();
        // test sekvenca
        long i = 0;
        for (i=0; i<1000000; i++) {
                analogRead(A0);
        } // kraj test sekvence
        long endTime= millis();
        Serial.println("Kraj testa");
        Serial.print("Trajanje u sekundama: ");
        Serial.println((endTime-startTime)/1000);
}
void loop() { }
```

Суштина ових измена се огледа у чињеници да је фреквенција одабирања сигнала повећана са 128 KHz на 1 MHz. Извршењем модификованог програма добија се резултат, приказан на слици 12.10, из којег се види да исти број конверзија захтева само 15 секунди за њихову реализацију (приближно 6,5 пута брже). При томе не треба заборавити ограничења у погледу меморијског простора за смештај добијених резултата. Меморијски простор унутар микроконтролера има RAM чији капацитет износи само 2KB.



Слика 12.10 Резултат извршења програма AnalogReadFast

12.5. Сажетак

Програмска решења треба да минимизирају време приступа расположивим периферијским јединицама и рачунско време неопходно за генерисање резултата обраде података у циљу генерисања управљања.

Уколико је прихватљиво извесно толеранцијско одступање, у погледу трајања неког процеса боље је уместо функције delay користити програмско решење којим се стално опсервира колико је времена протекло од почетка извршења конкретног програма (функције које нуде ову могућност millis и micros).

Управљање у реалном времену захтева да се рачунања изведу у што краћем времену. Кад год је могуће, треба користити целобројну аритметику јер се математичке операције тада извршавају брже.

Примена look-up табела омогућава значајне уштеде временских ресурса у случају потребе сукцесивног изтрачунавања низа вредности неке зависно променљиве, као на пример у случају синусне функције.

Операције очитавања и постављања логичких стања на пиновима (READ/WRITE операције) могу да се обаве значајно брже уколико се приступ конкретном пину оствари преко унутрашњих регистара микроконтролера.

Резултати аналогно дигиталне конверзије могу да се добију и више пута у времену уколико се уместо функције analogRead примене наредбе којим ће се променити режим рада одговарајућег тајмера и припадајућег прескалера. При томе треба имати у виду ограничења у погледу расположивог меморијског простора за прихват већег броја података (RAM меморија микроконтролера има капацитет од само 2KB).

Питања:

1. Како се може остварити мерење времена извршења делова неке програмске секвенце?
2. Које су предности и мане целобројне ариметике (integer) у односу на аритметику са пливајућим зарезом (float)?
3. На који начин се могу убрзати операције очитавања и постављања логичких стања на пиновима (READ/WRITE операције)?
4. Како се програмски може повећати број аналогнодигиталних конверзија у јединици времена?
5. Да ли се блок наредби извршава једнако брзо уколико се понавља унутар while циклуса који је смештан унутар void setup() функције и када се истоветан блок наредби понавља унутар void loop() функције?

ЛИТЕРАТУРА

[1] M. Rafiquzzaman: Fundamentals of Digital Logic anad Microcontrollers; Sixth Edition; ISBN 978-1-118-85579-9; Johm Wilez & Sons. 2014.

[2] Ph.D. Jack Purdum: Beginning C for ARDUINO, ISBN 978-1-4302-4777-7, Apres, 2012.

[3] Верица Васиљевић, Борислав Хаџибабић, Бранислав Павић, Владимир Тадић: МИКРОРАЧУНАРИ, Висока школа електротехнике и рачунарства струковних студија, Београд 2009.

[4] Laslo Kraus: Programski jezik C sa rešenim zadacima, ISBN 978-86-7466-511-4, Akademska misao, Београд 2014.

ИНДЕКС ПОЈМОВА

С

Т

У

Ф

Ц

Ш